Incremental Structural Clustering for Dynamic Networks

Yazhong Chen[1], Rong-Hua Li[1], Qiangqiang Dai[1], Zhenjun Li[1](✉), Shaojie Qiao[2], and Rui Mao[1]

[1] College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China
874908722@qq.com, rhli@szu.edu.cn, 1134133685@qq.com, 15323940@qq.com, mao@szu.edu.cn

[2] Chengdu University of Information Technology, Chengdu, China
qiaoshaojie@gmail.com

Abstract. Graph clustering is a fundamental tool for revealing cohesive structures in networks. The structural clustering algorithm for networks (SCAN) is an important approach for this task, which has attracted much attention in recent years. The SCAN algorithm can not only use to identify cohesive structures, but it is also able to detect outliers and hubs in a static network. Most real-life networks, however, frequently evolve over time. Unfortunately, the SCAN algorithm is very costly to handle such dynamic networks. In this paper, we propose an efficient incremental structural clustering algorithm for dynamic networks, called ISCAN. The ISCAN algorithm can efficiently maintain the clustering structures without recomputing the clusters from scratch. We conduct extensive experiments in eight large real-world networks. The results show that our algorithm is at least three orders of magnitude faster than the baseline algorithm.

1 Introduction

Network data are ubiquitous. Most real-world networks such as social networks, communication networks, and biological networks contain community structures. Discovering the community structures from a network is very useful for a number of applications. For example, in the biological network, a community may represent the molecule with common properties. In the communication network, a community may denote a close group which frequently communicate with each other.

Graph clustering is a fundamental tool to identify such community structures. In the last decade, there are a huge number of models and algorithms that have been proposed for graph clustering. A comprehensive survey on graph clustering and community detection algorithms can be found in [8]. Among all these algorithms, the structural graph clustering algorithm SCAN proposed in [23] is an notable algorithm which has been successfully used in many network analysis tasks [23]. Unlike many other graph clustering algorithms, the streaking

© Springer International Publishing AG 2017
A. Bouguettaya et al. (Eds.): WISE 2017, Part I, LNCS 10569, pp. 123–134, 2017.
DOI: 10.1007/978-3-319-68783-4_9

feature of SCAN is that it is not only able to detect the clusters of a network, but it can also be identify hubs and outliers.

The idea of the SCAN algorithm is similar to a density-based clustering algorithm DBSCAN, which has been widely used for clustering spatial data. Specifically, the SCAN algorithm first defines the structural similarity between two end vertices of each edge in the graph. If the structural similarity for an edge is no less than a given threshold ε, then this edge will be preserved. Otherwise, the algorithm can delete that edge. After this processing, the vertex in the remaining graph that has at least k neighbors is called a core vertex. Then, the algorithm uses the core vertices as seeds, and expands the clusters from the seeds by following the structural similarity edges (more details can be found in Sect. 2).

Unfortunately, the SCAN algorithm is tailored for static graph data. However, real-world networks typically evolve over time. The naive structural clustering algorithm to handle the dynamic networks is to recompute all clusters from scratch using the SCAN algorithm. Clearly, such a naive solution is very costly, as the time complexity of SCAN algorithm is $O(m^{1.5})$ (m denotes the number of edges of the graph), which is nonlinear with respective to the graph size [2].

To overcome this problem, we propose an efficient incremental structural clustering algorithm for dynamic networks, called ISCAN. The ISCAN algorithm can efficiently maintain the clusters generated by the SCAN algorithm without recomputing all the clusters. Specifically, when an edge updating (insertion or deletion), the ISCAN algorithm only works on a small number of edges (i.e., the edges that their structural similarity may update). The structural similarity of the edges may decrease and increase when an edge updating (see Sect. 3). When the structural similarity of an edge decreases, we may need to split the clusters. On the other hand, when the structural similarity of an edge increases, we may merge the clusters. In ISCAN, we propose a BFS-forest structure to maintain the clusters. Each BFS-tree represents a cluster. We also use a set Φ to maintain the set non-tree edges such that the structural similarity of these edges are larger than the threshold ε. When the algorithm splits a BFS-tree, we need to scan the set Φ to check whether the split tree can be merged again by an edge in Φ. We conduct extensive experiments in eight large real-world networks. The results show that the ISCAN algorithm is at least three orders of magnitude faster than the baseline algorithm.

The rest of this paper is organized as follow. In Sect. 2, we briefly introduce the SCAN algorithm. We propose the ISCAN algorithm in Sect. 3. The experimental results are reported in Sect. 4. We survey the related work and conclude this paper in Sects. 5 and 6 respectively.

2 Preliminaries

In this section, we briefly introduce several key concepts used in the SCAN algorithm [23]. Let $G = (V, E)$ be a graph, where V and E denote the set of vertices and edges respectively. The vertex neighborhood of a vertex $v \in V$ is defined as $\Gamma(v) \triangleq \{w \in V | (v, w) \in E\} \cup \{v\}$. The structural similarity between two end vertices of an edge (u, v) is defined as

$$\sigma(u,v) \triangleq \frac{|\Gamma(u) \cap \Gamma(v)|}{\sqrt{|\Gamma(u)||\Gamma(v)|}}. \tag{1}$$

If u and v are not end vertices of an edge, we define $\sigma(u, v) = 0$. In the SCAN algorithm, if $\sigma(u, v)$ is no less than a given parameter ε, the vertices u and v will be assigned into the same cluster. The ε-neighborhood of a node v is defined as

$$N_\varepsilon(v) \triangleq \{w \in \Gamma(v) | \sigma(w,v) \geq \varepsilon\}. \tag{2}$$

A vertex v is called a core vertex if and only if $|N_\varepsilon(v)| \geq \mu$, i.e., $CORE_{\varepsilon,\mu}(v) \Leftrightarrow |N_\varepsilon(v)| \geq \mu$. In the SCAN algorithm, if v is a core vertex and $u \in N_\varepsilon(v)$, u will be assigned to the cluster where v belongs to, and we call u is directly structural reachable from v (denoted by $DirREACH_{\varepsilon,\mu}(v, u)$). Formally, we define direct structure reachability as

$$DirREACH_{\varepsilon,\mu}(v,u) \Leftrightarrow CORE_{\varepsilon,\mu}(v) \wedge w \in N_\varepsilon(v) \tag{3}$$

If $DirREACH_{\varepsilon,\mu}(v, u)$ and $DirREACH_{\varepsilon,\mu}(u, w)$ hold, we call w is structural reachable from v (denoted by $REACH_{\varepsilon,\mu}(v, w)$). Formally, it is defined by

$$REACH_{\varepsilon,\mu}(v,w) \Leftrightarrow \exists v_1, ..., v_n \in V : v_1 = v \wedge v_n = w \wedge \forall i \in \{1, ..., n-1\} : DirREACH_{\varepsilon,\mu}(v_i, v_{i+1}). \tag{4}$$

If there exists a vertex $v \in V$ such that $REACH_{\varepsilon,\mu}(v, u)$ and $REACH_{\varepsilon,\mu}(v, w)$ hold, we call u and w meeting structure connectivity, denoted by $CONNECT_{\varepsilon,\mu}(u, w)$. Based on the above definitions, the cluster C in SCAN is defined as

Definition 1. $CLUSTER_{\varepsilon,\mu}(C) \Leftrightarrow$

(1) *Connectivity* : $\forall u, w \in C : CONNECT_{\varepsilon,\mu}(u, w)$
(2) *Maximality* : $\forall u, w \in V : u \in C \wedge REACH_{\varepsilon,\mu}(u, w) \Rightarrow w \in C$

The SCAN algorithm aims to find all clusters defined in Definition 1. Note that there may exist some vertices that do not belong to any cluster. Those vertices are considered as hubs if they bridge different clusters, otherwise they will be classified as outliers [23]. The SCAN algorithm first finds a core vertex, and then creates a new cluster for that core vertex. Then, the algorithm traverses the ε-neighborhood of the core vertex in a BFS (Breadth-first search) manner to add vertices into the cluster. When all the vertices are visited, the algorithm terminates. Note that the SCAN algorithm is tailored for static graphs, and it is nontrivial to maintain the clusters when the graphs evolve over time. In this paper, we focus on such a cluster maintenance problem when the graph is updated by an edge insertion and deletion.

3 Incremental Structure Clustering Algorithm

To maintain the clusters, a naive algorithm is to recompute all clusters by invoking SCAN when inserting or deleting an edge. Clearly, such a naive algorithm is inefficient. Below, we propose the ISCAN algorithm to maintain the clusters without recomputing all clusters. Our algorithm is based on the following key observations.

Observation 1. Consider an edge $e = (u, v)$. Let $N(e_{uv}) \triangleq \Gamma(u) \cup \Gamma(v)$, $R(e_{uv}) \subseteq E$ be the set of edges with two end vertices in $N(e_{uv})$. When insert or delete an edge $e = (u, v)$, we only need to update the structural similarity between the two end vertices of an edge in $R(e_{uv})$. There is no need to update the structural similarity between the two end vertices of an edge in $E \backslash R(e_{uv})$. When adding or removing an edge $e = (u, v)$, the structural similarity may increase or decrease for different edges in $R(e_{uv})$. Below, we focus mainly on the edge insertion case, and similar results also hold for the edge deletion case. When inserting an edge $e = (u, v)$, we have three different cases.

Algorithm 1. SCAN [23]

Input: G=(V,E), parameters ε and μ

Output: The BFS-forest and the non-tree-edge set Φ

```
 1: for all Unclassified vertex $v \in V$ do
 2:     if $CORE_{\varepsilon,\mu}(v)$ then
 3:         Generate new clusterID for $v$; Insert all $x \in N_\varepsilon(v)$ into queue $Q$;
 4:         $x.parent \leftarrow v$; {// create the tree strucuture}
 5:         while $Q \neq \emptyset$ do
 6:             $y \leftarrow Q.pop()$; $R$={$x \in V$ |$DirREACH_{\varepsilon,\mu}(y, x)$};
 7:             for all $x \in R$ do
 8:                 if $x$ is unclassified then
 9:                     Assign the current clusterID to $x$; Insert $x$ into queue $Q$;
10:                     $x.parent \leftarrow y$;
11:                 else
12:                     if $x.parent \neq y \wedge y.parent \neq x$ then Insert the edge $(y, x)$ into
    $\Phi$;
13:             Remove $y$ from $Q$;
```

First, the structural similarity between (u, v), i.e., $\sigma(u, v)$ increases to $\frac{|\Gamma(u) \cap \Gamma(v)|}{\sqrt{(|\Gamma(u)|+1)(|\Gamma(v)|+1)}}$ after inserting (u, v). Here $\Gamma(v)$ denotes the vertex neighborhood of v before inserting (u, v). This is because there is no edge between (u, v) before inserting (u, v), thus $\sigma(u, v) = 0$ before adding (u, v) by definition. Second, if (w, u, v) forms a triangle after inserting (u, v), $\sigma(w, v)$ will increase to $\frac{|\Gamma(w) \cap \Gamma(v)|+1}{\sqrt{|\Gamma(w)|(|\Gamma(v)|+1)}}$ based on the following lemma.

Lemma 1. $\frac{|\Gamma(w) \cap \Gamma(v)|}{\sqrt{|\Gamma(w)||\Gamma(v)|}} < \frac{|\Gamma(w) \cap \Gamma(v)|+1}{\sqrt{|\Gamma(w)|(|\Gamma(v)|+1)}}$

Proof. First , we have $\frac{|\Gamma(w) \cap \Gamma(v)|^2}{|\Gamma(w)||\Gamma(v)|} / \frac{(|\Gamma(w) \cap \Gamma(v)|+1)^2}{|\Gamma(w)|(|\Gamma(v)|+1)} = |\Gamma(w) \cap \Gamma(v)|^2(\sqrt{|\Gamma(v)|} + 1)/\sqrt{|\Gamma(v)|}(|\Gamma(w) \cap \Gamma(v)| + 1)^2$. Then, we have $|\Gamma(w) \cap \Gamma(v)|^2(\sqrt{|\Gamma(v)|} +$

$1)/\sqrt{|\Gamma(v)|}(|\Gamma(w)\cap\Gamma(v)|+1)^2 \leq |\Gamma(w)\cap\Gamma(v)|(\sqrt{|\Gamma(v)|}+1)/\sqrt{|\Gamma(v)|}(|\Gamma(w)\cap\Gamma(v)|+1)$. Since $|\Gamma(w)\cap\Gamma(v)| \leq |\Gamma(v)|$, we have $|\Gamma(w)\cap\Gamma(v)|(\sqrt{|\Gamma(v)|}+1)/\sqrt{|\Gamma(v)|}(|\Gamma(w)\cap\Gamma(v)|+1) \leq 1$. This completes the proof.

Third, if the vertices (w,u,v) do not form a triangle after adding (u,v), $\sigma(w,v)$ decreases to $\frac{|\Gamma(w)\cap\Gamma(v)|}{\sqrt{|\Gamma(w)|(|\Gamma(v)|+1)}}$. Based on this observation, when the structural similarity of (w,v) increases, we may merge two clusters. On the other hand, when the structural similarity decreases, we may need to split a cluster.

Observation 2. A crucial observation is that the clustering procedure of SCAN will generate a BFS-forest where each BFS-tree is a cluster [23]. Note that all the non-leaf nodes in a BFS-tree are the core vertices. Based on this, we can use the BFS-forest to maintain the clusters when the graph changes. In Algorithm 1, we give a modified SCAN algorithm to generate the BFS-forest (see lines 4 and 10).

3.1 The ISCAN Algorithm

As shown in Observation 1, each edge updating (inserting or deleting) can lead to the structural similarity decreasing or increasing. When the structural similarity of an edge (u,v) increases, the algorithm may need to merge the clusters of u and v if $u(v)$ is directly structural reachable from $v(u)$. Moreover, the vertices u and v may become core vertices, if they are not core before updating. On the other hand, if the structural similarity of (u,v) decreases, the algorithm may need to split the cluster, because $\sigma(u,v)$ may be smaller than the threshold ε. Also, the vertices u and v may become non-core vertices if they are core vertices before updating. The challenge is how can we maintain the BFS-forest structure to handle all these cases.

Algorithm 2. ISCAN

```
Input: $G=(V,E)$, $\varepsilon$ and $\mu$, the updated edge $e=(u,v)$, the non-tree-edge set $\Phi$
Output: The updated clusters
1: Let $N(e_{uv}) \triangleq \Gamma(u)\cup\Gamma(v)$, $R(e_{uv}) \subseteq E$ be the set of edges with two end vertices
   in $N(e_{uv})$;
2: Recompute the structural similarity for all edges in $R(e_{uv})$;
3: for all $w \in N(e_{uv})$ do
4:     if $w$ is a core vertex then MergeCluster($w$,$\Phi$);
5: for all $e=(\tilde{u},\tilde{v}) \in R(e_{uv})$ do
6:     if $\sigma(\tilde{u},\tilde{v}) \geq \varepsilon \wedge \sigma'(\tilde{u},\tilde{v}) < \varepsilon$ then SplitCluster($e$,$\Phi$);
7: for all $(\tilde{u},\tilde{v}) \in \Phi$ do
8:     if $\tilde{u}$.ClusterID $\neq$ $\tilde{v}$.ClusterID then Merge($\tilde{u}$,$\tilde{v}$); {// merge the trees that contain
   $w$ and $u$}
```

To tackle this challenge, we additionally maintain a set Φ which stores all the non-tree edges (u,v) such that $v(u)$ is directly structural reachable from u (v). Recall that by the SCAN algorithm, there may exist an edge (u,v) meeting the

$DirREACH$ relationship, i.e., $v(u)$ is directly structural reachable from $u(v)$ and (u, v) is not in any BFS-tree. We make use of the set Φ to keep all these edges. In other words, we classify the edges that satisfy the $DirREACH$ relationship into two classes: tree edge which is stored in the BFS-forest, and non-tree edge which is kept in Φ. When we split a BFS-tree into two sub-treess, we need to scan Φ to check whether these sub-trees can be merged again by an edge in Φ. The ISCAN algorithm maintains both the BFS-forest structure and the set Φ. Initially, we can obtain Φ using the modified SCAN algorithm as shown in Algorithm 1 (see line 12).

The ISCAN algorithm is outlined in Algorithm 2. It consists of three steps to maintain the clusters after an edge (u, v) updating. In the first step, the algorithm considers the case of structural similarity increasing. In this case, the algorithm scans the core vertices to maintain the BFS-forest and Φ. The algorithm recomputes the structural similarity for each edge in $R(e_{uv})$, because the structural similarity for these edges may be updated. For each core vertex in $N(e_{uv})$, the algorithm invokes Algorithm 3 to maintain the set Φ and merge the clusters (lines 1–4).

Algorithm 3. MergeCluster(w,Φ)

```
1:  if $w$ is unclassified then Create a new clusterID for $w$;
2:  for all $u \in N_\varepsilon(w)$ do
3:      if $u$ is classified then
4:          if $u$ is non-core vertex then
5:              if $(u.clusterID = w.clusterID \wedge u.parent \neq w) \vee$
6:                  $(u.clusterID \neq w.clusterID)$ then
7:                  Insert edge $(w, u)$ into $\Phi$;
8:          else
9:              if $u.clusterID = w.clusterID \wedge u.parent \neq w \wedge w.parent \neq u$ then
10:                 Insert edge $(w, u)$ into $\Phi$;
11:             if $u.clusterID \neq w.clusterID$ then
12:                 Merge($w$,$u$); {// merge the trees that contain $w$ and $u$}
13:     else
14:         $u.clusterID \leftarrow w.clusterID$; $u.parent = w$;
```

In Algorithm 3, the algorithm first checks whether the core vertex w is classified or not. If it is unclassified (i.e., w does not belong to any cluster), we create a cluster ID for w. Then, the algorithm traverses the ε-neighborhood of w. For each neighbor u in $N_\varepsilon(w)$, if u is unclassified, then we add u into the same cluster as w, and set w as the parent for u (line 13). Otherwise, the algorithm checks whether u is a core vertex. If that is the case, the algorithm verify whether (w, u) is a tree edge. If it is not a tree edge and w and u have the same cluster ID, we insert (w, u) into Φ (lines 8–9). If w and u have different cluster IDs, we merge the two trees (i.e., clusters) of w and u (line 10–11). On the other hand, if u is not a core vertex, we consider two cases. First, if (w, u) is not a tree edge and w, u have the same cluster ID, we insert (w, u) into Φ. Second, if (w, u) have different cluster IDs, we also add (w, u) into Φ (lines 4–6). For this case, we will add u into the cluster of w in the third step.

Algorithm 4. SplitCluster($e = (u, v)$,Φ)

```
1: if both $u$ and $v$ are core vertices then
2:     if $u.parent \neq v.parent \vee v.parent \neq u.parent$ then
3:         Delete $(u, v)$ from $\Phi$;
4:     else
5:         Remove $(u, v)$ from the BFS-tree.
6: if $u$ is core and $v$ is not core then
7:     if $v.parent = u$ then
8:         Remove $(u, v)$ from the BFS-tree.
9:     else
10:        Remove $(u, v)$ from $\Phi$;
11: if $v$ is core and $u$ is not core then
12:     Similar processing as the case of "$u$ is core and $v$ is not core";
13: if both $u$ and $v$ are not core vertices then
14:     if $u$ is core before updating then
15:         if $v.parent = u$ then
16:             Remove $(u, v)$ from the BFS-tree.
17:         else
18:             Remove $(u, v)$ from $\Phi$;
19:     if $v$ is core before updating then
20:         Similar processing as the case of "$u$ is core before updating";
```

In the second step, Algorithm 2 considers the case of when the structural similarity decreases. To this end, Algorithm 2 scans all the edges in $R(e_{uv})$. For an edge $e = (\tilde{u}, \tilde{v})$, if the structural similarity for e before updating (denoted by $\sigma(\tilde{u}, \tilde{v})$) is no less than ε and the structural similarity for e after updating (denoted by $\sigma'(\tilde{u}, \tilde{v})$) is smaller than ε, the algorithm invokes Algorithm 4 to split the BFS-trees and also maintain the set Φ.

In Algorithm 4, we consider four different cases for the input edge (u, v). First, both u and v are core vertices after updating. In this case, if (u, v) is not a tree edge, we delete (u, v) from Φ (lines 2–3). Otherwise, we remove (u, v) from the corresponding BFS-tree (line 5). Second, u is a core vertex and v is not. In this case, if u is a parent of v before updating, we remove (u, v) from the corresponding BFS-tree (lines 7–8). Otherwise, we remove it from Φ (line 10). Third, v is a core vertex, but u is not. This case is similar to the second case, thus we omit the details. Fourth, both u and v are not core vertices. In this case, we need to consider whether $u(v)$ is core before updating. If both u and v are not core vertices before updating, we do nothing. If u (v) is core and (u, v) is a tree edge, we delete (u, v) from the BFS-tree (lines 14–16). Otherwise, delete (u, v) from Φ.

In the third step, Algorithm 2 scans each edge $(\tilde{u}, \tilde{v})$ in Φ, and merge two clusters by the edge $(\tilde{u}, \tilde{v})$ if $\tilde{u}$ and $\tilde{v}$ have different cluster IDs. Since the ISCAN algorithm enumerates all the possible cases for updating both the BFS-forest and Φ, it is correct. Below, we analyze the time and space complexity of the algorithm.

Complexity Analysis. We first analyze the time complexity of the ISCAN algorithm. Let m and n be the number of edges and vertices of the graph G respectively. Let $\tilde{m} = |\Phi|$ be the size of Φ. Clearly, $\tilde{m}$ is much smaller than m in real-world graphs. In our experiments, we show that in the Youtube social network $m = 2,987,624$ whereas $\tilde{m} = 3,210$. Initially, the algorithm recomputes the structural similarity for all edge in $R(e_{uv})$. Let $O(T)$ be the time spent in this initial step. Since $|R(e_{uv})|$ is very small, $O(T)$ typically can be dominated by $O(m)$ in real-world graphs. In the first step, the cluster merging procedure can be done in $O(n)$ time, because in the worst case, we merge at most $O(n)$ trees. In the second step, we also at most split $O(n)$ clusters, thus the time spent in this step can be bounded by $O(n)$. In the last step, the algorithm takes $O(\tilde{m})$ time to scan Φ and merge the clusters. Putting it all together, we can conclude that the time complexity is $O(m + n)$. In the experiments, we will show that the time usage of our algorithm is much less than such a worst case bound. For the space complexity, our algorithm only need to maintain the BFS-forest and Φ which is dominated by $O(m + n)$.

4 Performance Studies

In this section, we conduct extensive experiments to evaluate the performance of the proposed algorithm. We implement two algorithms: ISCAN and Basic. The ISCAN algorithm is the proposed algorithm, while the Basic algorithm recomputes the clustering results using the SCAN algorithm when the graph changes. We implement these algorithms in C++. All the experiments are conducted in a Linux Server with 2 CPUs and 32 GB main memory.

Dataset. We use four real-world large datasets in the experiments. The detailed statistics of the datasets are summarized in Table 1. All these datasets are downloaded from (http://konect.uni-koblenz.de/networks/). The first three datasets (Youtube, Pokec, and Flixster) are social networks, and the following three datasets (WebGoogle, WebBerkStan, and TREC) are web graphs. The Skitter dataset is a computer network and the RoadNetPA dataset is a road network.

Parameter Setting. There are two parameters in our algorithm: ε, and μ. As recommended in [23], we set the default values of ε and μ by 0.5 and 2, respectively. We vary ε from 0.3 to 0.8, and vary μ from 2 to 7. In all experiments, when varying a parameter, we set the default value for the other parameter. In all experiments, we randomly insert and delete 1000 edges from the original network. For each edge update, we invoke the ISCAN and Basic algorithm to update the clustering results. We record the total time for each algorithm to handle the 1000 edge insertions and deletions.

Efficiency Testing (vary ε). In this experiment, we evaluate the efficiency of our algorithm when varying ε. The results are shown in Fig. 1. As can be seen, the ISCAN algorithm is at least three orders of magnitude faster than the Basic algorithm over all the datasets. For example, in Youtube dataset, when $\varepsilon = 0.5$, our algorithm takes only 10 s to update 1000 edges, whereas the

Table 1. Datasets

Datasets	Number of nodes	Number of edges
Youtube	1,134,000	2,987,000
Pokec	1,632,000	22,301,000
Flixster	2,523,000	7,918,000
WebGoogle	875,000	4,322,000
WebBerkStan	685,000	6,649,000
TREC	1,601,000	6,679,000
Skitter	1,696,000	11,095,000
RoadNetPA	1,088,000	1,541,000

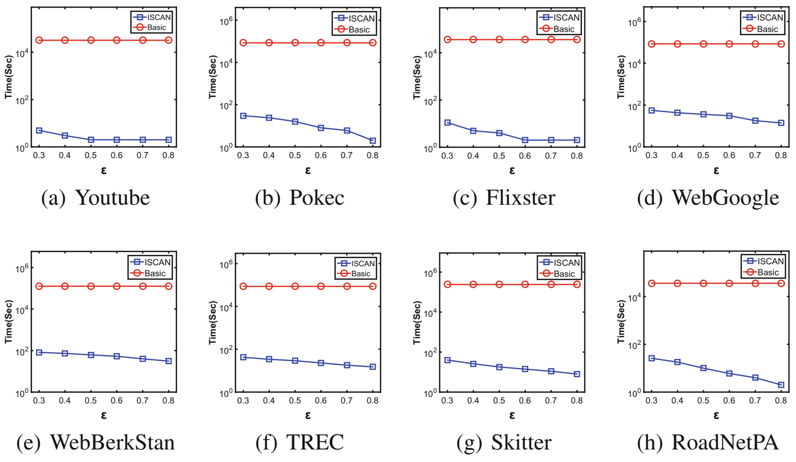
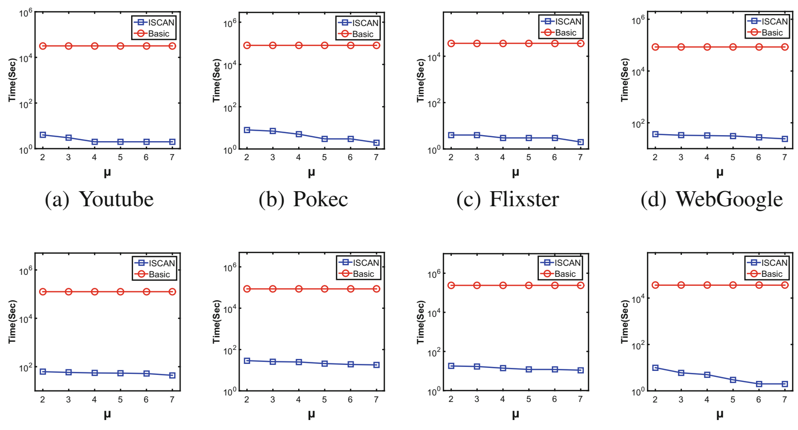


Fig. 1. Comparison between ISCAN and Basic (vary ε)

Basic algorithm consumes more than 10000 s. Moreover, the running time of our algorithm generally decreases with increasing ε, while the running time of Basic keeps stable with varying ε. The reason is as follows. When ε is large, the clusters obtained by the SCAN algorithm are relatively stable with respect to an edge updating. As a result, our algorithm may only need to update a small amount of edges. For the Basic algorithm, the algorithm always invoke SCAN to recompute the clusters, thus its running time is insensitive to an edge updating.

Efficiency Testing (vary μ). In this experiment, we compare the efficiency between ISCAN and Basic when varying μ. The results are reported in Fig. 2. From Fig. 2, we can see that the ISCAN algorithm is at least three orders of magnitude faster than the Basic algorithm with different μ values in all datasets. Furthermore, the running time of ISCAN decreases as μ increase. The rationale is as follow. When the graph updating, the larger value of μ, the less influence for the original clusters. Therefore, our algorithm is more efficient when μ is large.



(a) Youtube (b) Pokec (c) Flixster (d) WebGoogle

(e) WebBerkStan (f) TREC (g) Skitter (h) RoadNetPA

Fig. 2. Comparison between ISCAN and Basic (vary μ)

Similarly, for the Basic algorithm, it is robust with respect to the parameter μ, as it always recompute the clusters using the SCAN algorithm.

To summarize, we can conclude that the ISCAN algorithm is very efficient in practice. As shown in Figs. 1 and 2, under the default parameter setting, the ISCAN algorithm takes only a few seconds to update the clusters in a large graph (e.g., in Pokec dataset, it has more than 22 million edges) with 1000 edge updates. These results demonstrate the high efficiency of the proposed algorithm.

5 Related Work

Structural Graph Clustering. The original structural graph clustering algorithm (SCAN) was proposed by Xu et al. in [23]. Recently, Shiokawa et al. [20] proposed an improved SCAN algorithm called SCAN ++. The SCAN ++ algorithm is based on a new data structure, called directly two-hop-away reachable node set (DTAR). Specifically, DTAR maintains the set of two-hop-away nodes from a given node which are likely to be in the same cluster as the given node. To further reduce the running time of the SCAN algorithm, Chang et al. [2] developed a two-step algorithm called pSCAN. The pSCAN algorithm first clusters the core nodes, and then clusters the border nodes. They also proposed an efficient technique to cluster the core nodes based on a union-find structure. All those SCAN algorithms are tailored for the static graphs, and they are costly to handle the dynamic graphs.

Cohesive Subgraph and Community Detection. Our work is closely related to the cohesive subgraph detection problem which aims to find the densely connected subgraphs from a graph. There are a number of cohesive subgraph models proposed in the literature. Notable examples consist of the maximal clique [4], k-core [12,15,24], k-truss [5,21], maximal k-edge connected subgraph (MkCS) [1,3,25], locally dense subgraph [14], influential community [10,11], and so on.

All those methods can be used to find the non-overlapped communities, and a comprehensive survey on the other community detection algorithms can be found in [8]. Another line of studies focus on finding overlapped communities. For example, Cui et al. [6] proposed an α-adjacency γ-quasi-k-clique model to study the problem of overlapped community search. More recently, Huang et al. [9] introduce a k-truss community model to detect overlapped communities. An excellent survey on overlapped community detection can be found in [22].

Community Maintenance in Dynamic Networks. The community maintenance problem in dynamic networks is an important task in social network analysis [7]. Our work is also closely related to this issue. For the community maintenance problem, it is very often not necessary to recompute the communities when the graph changes. One can only need to detect the affected edges or nodes in a community after the the graph updating. Clearly, different community models have different community updating strategies. Notable community updating algorithms are listed as follows. For the maximal clique model, Cheng et al. [4] introduced an algorithm for dynamically updating the maximal cliques in massive networks. For the k-core model, Li [12] proposed an efficient core maintenance in large dynamic graphs. Similarly, for the k-truss model, Huang [9] proposed an efficient truss maintenance algorithm for dynamic networks. Different from all the existing algorithm, in this paper, we study the problem of dynamically updating the clustering results generated by the SCAN algorithm. Our algorithms may also work on location-based social networks [?], spatial networks [17,19] and trajectory data [16,18]. In the next step, we will study dynamic algorithms in the metric space [13].

6 Conclusion

In this paper, we study the incremental structural clustering problem for dynamic network data. We propose a new algorithm called ISCAN to efficiently maintain the clusters generated by the SCAN algorithm. In the ISCAN algorithm, we use a BFS-forest and a non-tree edge set structure to maintain the clusters. We conduct comprehensive experiments over eight large real-world networks, and the results demonstrate the high efficiency of our algorithm.

Acknowledgement. We thank anonymous reviewers for their insightful comments. The work was supported in part by NSFC Grants (61402292, U1301252, 61033009), NSF-Shenzhen Grants (JCYJ20150324140036826, JCYJ20140418095735561), and Startup Grant of Shenzhen Kongque Program (827/000065).

References

1. Akiba, T., Iwata, Y., Yoshida, Y.: Linear-time enumeration of maximal k-edge-connected subgraphs in large networks by random contraction. In: CIKM (2013)
2. Chang, L., Li, W., Lin, X., Qin, L., Zhang, W.: pSCAN: Fast and exact structural graph clustering. In: ICDE, pp. 253–264 (2016)

3. Chang, L., Yu, J.X., Qin, L., Lin, X., Liu, C., Liang, W.: Efficiently computing k-edge connected components via graph decomposition. In: SIGMOD (2013)
4. Cheng, J., Ke, Y., Fu, A.W.C., Yu, J.X., Zhu, L.: Finding maximal cliques in massive networks. ACM Trans. Database Syst. **36**(4), 21 (2011)
5. Cohen, J.: Trusses: cohesive subgraphs for social network analysis. Technique report (2005)
6. Cui, W., Xiao, Y., Wang, H., Lu, Y., Wang, W.: Online search of overlapping communities. In: SIGMOD (2013)
7. Eppstein, D., Galil, Z., Italiano, G.F.: Dynamic graph algorithms. In: Algorithms and theory of computation handbook, p. 9 (1996)
8. Fortunato, S.: Community detection in graphs. Phys. Rep. **486**(3–5), 75–174 (2010)
9. Huang, X., Cheng, H., Qin, L., Tian, W., Yu, J.X.: Querying k-truss community in large and dynamic graphs. In: SIGMOD (2014)
10. Li, R., Qin, L., Yu, J.X., Mao, R.: Influential community search in large networks. PVLDB **8**(5), 509–520 (2015)
11. Li, R., Qin, L., Yu, J.X., Mao, R.: Finding influential communities in massive networks. VLDB J. (2017)
12. Li, R., Yu, J.X., Mao, R.: Efficient core maintenance in large dynamic graphs. IEEE Trans. Knowl. Data Eng. **26**(10), 2453–2465 (2014)
13. Mao, R., Zhang, P., Li, X., Liu, X., Lu, M.: Pivot selection for metric-space indexing. Int. J. Mach. Learn. Cybern. **7**(2), 311–323 (2016)
14. Qin, L., Li, R., Chang, L., Zhang, C.: Locally densest subgraph discovery. In: KDD (2015)
15. Seidman, S.B.: Network structure and minimum degree. Soc. Netw. **5**(3), 269–287 (1983)
16. Shang, S., Chen, L., Jensen, C.S., Wen, J.R., Kalnis, P.: Searching trajectories by regions of interest. IEEE Trans. Knowl. Data Eng. **99**, 1–1 (2017)
17. Shang, S., Chen, L., Wei, Z., Jensen, C.S., Wen, J.R., Kalnis, P.: Collective travel planning in spatial networks. In: IEEE International Conference on Data Engineering, pp. 59–60 (2017)
18. Shang, S., Ding, R., Yuan, B., Xie, K., Zheng, K., Kalnis, P.: User oriented trajectory search for trip recommendation. In: EDBT, pp. 156–167 (2012)
19. Shang, S., Ding, R., Zheng, K., Jensen, C.S., Kalnis, P., Zhou, X.: Personalized trajectory matching in spatial networks. VLDBJ **23**(3), 449–468 (2014)
20. Shiokawa, H., Fujiwara, Y., Onizuka, M.: Scan++: efficient algorithm for finding clusters, hubs and outliers on large-scale graphs. PVLDB **8**(11), 1178–1189 (2015)
21. Wang, J., Cheng, J.: Truss decomposition in massive networks. PVLDB **5**(9), 812–823 (2012)
22. Xie, J., Kelley, S., Szymanski, B.K.: Overlapping community detection in networks: the state-of-the-art and comparative study. Acm Comput. Surv. **45**(4), 43 (2011)
23. Xu, X., Yuruk, N., Feng, Z., Schweiger, T.A.J.: Scan: a structural clustering algorithm for networks. In: KDD, pp. 824–833 (2007)
24. Zheng, D., Liu, J., Li, R., Aslay, Ç., Chen, Y., Huang, X.: Querying intimate-core groups in weighted graphs. In: 11th IEEE International Conference on Semantic Computing, ICSC (2017)
25. Zhou, R., Liu, C., Yu, J.X., Liang, W., Chen, B., Li, J.: Finding maximal k-edge-connected subgraphs from a large graph. In: EDBT (2012)