Efficient Community Search over Large Directed Graphs: An Augmented Index-based Approach

Yankai Chen[1] , **Jie Zhang**[2] , **Yixiang Fang**[3*] , **Xin Cao**[3] and **Irwin King**[1]
[1]Department of Computer Science and Engineering, The Chinese University of Hong Kong
[2]School of Computer Science and Engineering, Nanyang Technological University
[3]School of Computer Science and Engineering, The University of New South Wales
{ykchen, king}@cse.cuhk.edu.hk, zhangj@ntu.edu.sg, {yixiang.fang, xin.cao}@unsw.edu.au

Abstract

Given a graph G and a query vertex q, the topic of *community search* (CS), aiming to retrieve a dense subgraph of G containing q, has gained much attention. Most existing works focus on undirected graphs which overlooks the rich information carried by the edge directions. Recently, the problem of community search over directed graphs (or CSD problem) has been studied [Fang *et al.*, 2019b]; it finds a connected subgraph containing q, where the in-degree and out-degree of each vertex within the subgraph are at least k and l, respectively. However, existing solutions are inefficient, especially on large graphs. To tackle this issue, in this paper we propose a novel index called *D-Forest*, which allows a CSD query to be completed within the optimal time cost. We further propose efficient index construction methods. Extensive experiments on six real large graphs show that our index-based query algorithm is up to two orders of magnitude faster than existing solutions.

1 Introduction

With the rapid development of information technologies, large graphs are ubiquitous in various areas (e.g., social networks and biological science) [Li *et al.*, 2015b; Hu *et al.*, 2017; Zhu *et al.*, 2019; Hu *et al.*, 2019; Wan *et al.*, 2020]. Finding communities over these graphs is fundamental to many real applications, such as event organization, recommendation, and network analysis. In recent years, the topic of *community search* (CS) has gained much attention (e.g., [Sozio and Gionis, 2010; Cui *et al.*, 2014; Huang *et al.*, 2014; Fang *et al.*, 2019c; Fang *et al.*, 2019a]), which aims to find dense communities containing the query vertex q from a graph G in an online manner.

Earlier CS works (e.g., [Sozio and Gionis, 2010; Cui *et al.*, 2013; Cui *et al.*, 2014]) mainly focus on undirected graphs, where the graph edges do not have directions. They often require the community to be a connected subgraph satisfying a particular metric of structure cohesiveness (e.g., each vertex within the community has a degree of k or more). Later on, the attributes (e.g., influence values [Li *et al.*, 2015a], keywords [Fang *et al.*, 2017a], locations [Fang *et al.*, 2017b], and profiles [Chen *et al.*, 2019]) of graphs have also been considered for CS. However, all these works ignore the directions of edges. As pointed out in [Malliaros and Vazirgiannis, 2013; Zhang *et al.*, 2014], the ignorance of directions of edges, failing to capture asymmetric relationships implied by the directions, may lead to noise and inaccurate results.

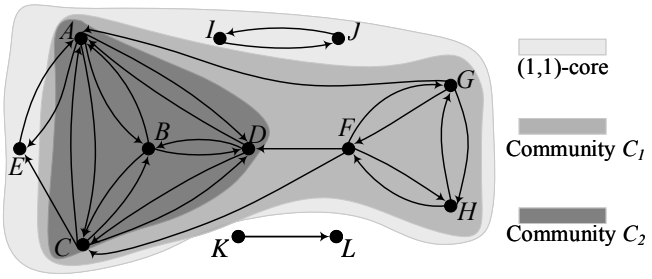

Figure 1: A directed graph.

To remedy the issue above, Fang et al. [2019b] studied the problem of CS over directed graphs, or CSD problem. Specifically, given a query vertex q of a directed graph G, and two integers k and l, it aims to find a maximum connected subgraph containing q, where the in-degree and out-degree of each vertex within the subgraph are at least k and l, respectively. For example, in Figure 1, let q=B and k=l=2; then, the subgraph C_1 will be returned; if k=l=3, then the subgraph C_2 is the answer. The minimum degree constraints have also been used in the (k, l)-core [Giatsidis *et al.*, 2011], or the maximum subgraph in which each vertex's in-degree and out-degree are at least k and l, respectively. Note that the (k, l)-core may not be a connected subgraph.

To answer a CSD query, an online method is to iteratively peel vertices that do not satisfy the degree constraints until finding the subgraph satisfying both the connectivity and minimum degree constraints. Obviously, this iterative approach could be costly. To improve efficiency, Fang et al. [2019b] develop indexes, which pre-compute all the (k, l)-cores and then organize them compactly. Given a CSD query, they first find the (k, l)-core using the indexes, and then compute the maximum connected subgraph containing q from the (k, l)-core. Clearly, since this maximum connected subgraph is of-

*Corresponding author

ten much smaller than the (k, l)-core, their solutions are inefficient if the (k, l)-core is very large. For example, on a graph with about 4 billion edges, they may take over 33 minutes to answer 200 queries as shown by our experiments.

To facilitate efficient CSD queries, in this paper we develop a novel index structure, called D-Forest, which not only compactly organizes all the (k, l)-cores, but also takes the connectivity into consideration. Specifically, we first compute all the (k, l)-cores sequentially where k and l range from 0 to their maximum values. Then, for each value of k, we compute their connected components and organize them into a tree structure, where each tree node corresponds to a connected (k, l)-core. As a result, the index is a forest, consisting of a list of trees. Given the D-Forest, to answer a CSD query, we first find the k-th tree and then return the connected (k, l)-core containing q. Clearly, the query takes the optimal query time cost, i.e., $O(|C|)$, where C is the set of vertices in the community. Experiments on six real large graphs show that our index construction process takes comparable time cost with those of existing algorithms, and our query algorithm takes only 14.4 seconds to answer 200 queries on a graph with around 4 billion edges.

In summary, our main contributions are as follows:

- We develop a novel index D-Forest, based on which a CSD query can be completed in optimal time cost.
- To build the D-Forest, we propose a basic algorithm, and an advanced algorithm by introducing an auxiliary data structure called Core-based Union-Find (or CUF).
- We perform extensive experiments on six real large graphs; results show that our query algorithm is up to two orders of magnitude faster than existing solutions.

2 Related Works

Community detection (CD). Generally, CD aims to detect all the communities from an entire graph. Earlier classic solutions (e.g., [Fortunato, 2010; Newman and Girvan, 2004]) rely on edge-based analysis (e.g., modularity maximization) to discover these communities. However, most of them focus on undirected graphs. Recent works start to detect communities from directed graphs. In [Leicht and Newman, 2008; Kim *et al.*, 2010], the concept of modularity maximization [Newman and Girvan, 2004] is extended for CD on directed graphs. In [Lancichinetti and Fortunato, 2009], authors introduced new benchmark graphs to test CD methods over directed graphs. Yang et al. [2010] introduced a new stochastic block model called PPL to find communities in directed graphs; they also detected overlapped communities in directed graphs [Yang *et al.*, 2014]. Besides, there are also some local CD methods (e.g., [Flake *et al.*, 2000; Ning *et al.*, 2016]). A recent survey of CD solutions on directed graphs can be found in [Malliaros and Vazirgiannis, 2013]. However, these CD methods are often time consuming, especially on large graphs, and also it is not clear how they can be adapted for online CS.

Community search (CS). CS finds the community from a large graph in a fast and online manner, based on a query request. To measure the structure cohesiveness of communities, the k-core metric is often employed, requiring that each vertex of the community should have a degree of k or more, where k is a given integer [Batagelj and Zaversnik, 2003; Sozio and Gionis, 2010; Cui *et al.*, 2014; Li *et al.*, 2015a; Fang *et al.*, 2017a; Fang *et al.*, 2017c; Fang *et al.*, 2017b; Chen *et al.*, 2019; Wang and Zhu, 2019; Fang *et al.*, 2019d; Fang *et al.*, 2020]. Other cohesiveness metrics have also been considered for CS, such as k-clique [Cui *et al.*, 2013], k-truss [Huang *et al.*, 2014; Huang *et al.*, 2015; Huang and Lakshmanan, 2017; Ebadian and Huang, 2019] and k-ECC [Hu *et al.*, 2017], pagerank-based [Andersen and Lang, 2006], etc. A survey of CS over graphs can be found in [Fang *et al.*, 2019a]. However, most of these works focus on undirected graphs. A recent work [Fang *et al.*, 2019b] has studied CS over directed graphs, but its solutions are still inefficient for large graphs, calling for more efficient CS approaches.

3 Problem Definition

We consider a directed graph $G(V, E)$ with a vertex set V and an edge set E. The sizes of V and E are respectively denoted by n and m. The in-degree and out-degree of a vertex v in G are denoted by $deg_G^{in}(v)$ and $deg_G^{out}(v)$. Next, we introduce the core model on directed graphs.

Definition 1 ((k, l)-core [Giatsidis *et al.*, 2011])**.** *Given a directed graph $G(V, E)$ and two non-negative integers k and l, the (k,l)-core of G is the largest subgraph G' of G, such that $\forall v \in G'$, $deg_{G'}^{in}(v) \geq k$ and $deg_{G'}^{out}(v) \geq l$.*

In the (k, l)-core, each vertex has at least k in-neighbours and l out-neighbours, so it is well engaged in the subgraph especially when k and l are large. This implies that the (k, l)-core is a cohesive subgraph, and thus can be used to model the cohesiveness of the community [Fang *et al.*, 2019b]. However, the (k, l)-core may not be a connected subgraph, so the connectivity constraint should be further imposed to model the community. Note that for simplicity, we denote a connected (k, l)-core by (k, l)-$\widehat{core}$.

Based on the discussions above, Fang et al. [2019b] formally introduced the problem of Community Search over Directed graphs (CSD) problem:

Problem 1 (CSD problem [Fang *et al.*, 2019b])**.** *Given a directed graph $G(V, E)$, a query vertex q, and two positive integers k and l, return the (k, l)-$\widehat{core}$ containing q.*

For example, in Figure 1, the (1, 1)-core, (2, 2)-core, and (3, 3)-core are marked in three different colors, where the (1, 1)-core has three connected components. If q=B and k=l=3, then the (3, 3)-$\widehat{core}$ C_2 is returned as the community.

4 Our Index-based Approach

To enable efficient CSD queries, in this paper we propose a novel index, called D-Forest, which allows the targeted community to be retrieved directly without examining the (k, l)-core. As a result, the query time cost is optimal. Meanwhile, the index is space efficient since it takes $O(m)$ space cost. In the following sections, we first give an overview of D-Forest, and then present two algorithms to build D-Forest.

4.1 Index Overview

We begin with an interesting lemma.

Lemma 1 ([Fang *et al.*, 2019b]). *Given a directed graph G, for any (k, l)-core with $l>0$, it is a subgraph of the $(k, l-1)$-core, i.e., the (k, l)-core is nested within the $(k, l-1)$-core.*

By Lemma 1, we can conclude that for any specific value of k, all the (k, l)-cores where l ranges from 0 to its maximum value can be organized into a chain such that each one is nested within its previous one. Similarly, the nested property above holds for the connected components of (k, l)-cores; that is, for any (k, l)-$\widehat{core}$ with $l>0$, it is nested within a particular $(k, l-1)$-$\widehat{core}$, so we can get a chain for each (k, l)-$\widehat{core}$. Consequently, for each value of k, we can build a tree structure called k-tree, by hierarchically organizing all these chains, such that each subtree corresponds to a (k, l)-$\widehat{core}$.

For example, all (k, l)-$\widehat{core}$s in Figure 1 can be organized into 4 trees, as depicted in Figure 2, where the k-tree is built for the value of k and each subtree contains all the vertices of a particular (k, l)-$\widehat{core}$. For instance, in the 1-tree, the subtree rooted at node [1] p, as shown in the dashed box, contains vertices $\{F, G, H\}$ and $\{A, B, C, D\}$, which are the vertices in the $(1, 2)$-$\widehat{core}$. Note that the number attached in each node indicates the value of l for the corresponding (k, l)-$\widehat{core}$ and we use the root node t to keep the tree shape.

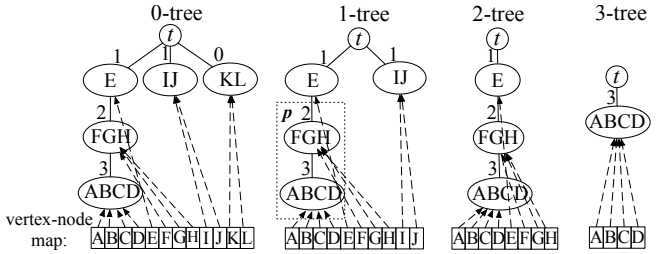

Figure 2: An example D-Forest index.

To summarize, in the k-tree, each node has four elements: (1) $parent$: a pointer to its parent node; (2) $childList$: a list of pointers to its child nodes; (3) $vSet$: a set of vertices, which are in the (k, l)-$\widehat{core}$ but not in the $(k, l-1)$-$\widehat{core}$; (4) $coreNum$: the value of l of the (k, l)-$\widehat{core}$, which corresponds to the subtree rooted at this node.

To enable efficient locating of the (k, l)-$\widehat{core}$, for each tree, we build an auxiliary map, where each key is a vertex and its value points to the node containing the vertex in the tree.

Lemma 2 (Space cost). *Given a directed graph G, its D-Forest takes $O(m)$ space.*

Proof. For each vertex v, if its in-degree is k, then it appears in at most k trees, and in each tree, it appears only twice (one in the tree and one in the auxiliary map). Thus, the space cost of v is bounded by $O(deg_G^{in}(v))$. Hence, Lemma 5 holds. □

Query algorithm. To answer a CSD query, we can first use the auxiliary map to locate the tree node p which contains q, then find the root of the subtree corresponding to the (k, l)-$\widehat{core}$ containing q, and finally return the set C of vertices in the subtree. We denote the query algorithm above by `IDX-Q`.

Lemma 3 (Query cost). *Given a D-Forest,* `IDX-Q` *completes in the optimal time and space cost, i.e., $O(|C|)$.*

Proof. The lemma directly follows the observation. □

Next, we present two index construction algorithms, which work in the top-down and bottom-up manners, respectively.

4.2 A Top-Down Index Construction Method

In [Fang *et al.*, 2019b], an efficient algorithm of decomposing (k, l)-cores for a graph is developed. It enumerates k from 0 to its maximum value $k_{\max}$, and for each k, it computes all the (k, l)-cores where l ranges from 0 to its maximum value $l_{\max}$. Since D-Forest is comprised of $(k_{\max}+1)$ trees, we can also enumerate k from 0 to $k_{\max}$, and for each specific k, we build the k-tree by the following the steps:

1. compute the $(k, 0)$-core, create a node with a set of vertices in $(k, 0)$-core, and initialize l=1;
2. compute all the (k, l)-$\widehat{core}$s from the $(k, l-1)$-$\widehat{core}$s;
3. for each (k, l)-$\widehat{core}$, create a node p ($p.vSet$ contains vertices in (k, l)-$\widehat{core}$, $p.coreNum$=l), link p to its parent node p', and update $p'.vSet$ as $p'.vSet \backslash p.vSet$;
4. increase l by 1, and repeat steps 2 and 3 until l reaches its maximum value.

Since the above method builds trees of D-Forest in a top-down manner, we denote it by `TopDown`.

Lemma 4. *Given a directed graph G, the time cost of building D-Forest using* `TopDown` *is $O(m^2)$.*

Proof. Given a specific k, computing all the (k, l)-cores from the $(k, 0)$-core takes $O(m)$ time; besides, searching all the (k, l)-$\widehat{core}$s from the $(k, l-1)$-$\widehat{core}$s takes $O(m)$ [Fang *et al.*, 2019b]. Thus, it takes $O(l_{\max} \cdot m)$ time to build the k-tree. Since there are $(k_{\max}+1)$ trees, the total cost is $O(k_{\max} \cdot l_{\max} \cdot m)$. Meanwhile, $k_{\max}$ and $l_{\max}$ are at most $(\sqrt{4m+1}-1)/2$ [Fang *et al.*, 2019b], so Lemma 4 holds. □

4.3 A Bottom-Up Index Construction Method

While `TopDown` is easy to implement, it may suffer from the low efficiency issue, as shown in Lemma 4. To further improve the efficiency, we propose another more efficient index construction method `BottomUp`, by introducing an auxiliary data structure called Core-based Union-Find (or CUF), which builds the trees in a bottom-up manner. Next, we first give an overview of `BottomUp`, and then introduce the details.

Overview of `BottomUp`. Unlike `TopDown`, `BottomUp` enumerates the values of k from $k_{\max}$ to 0, and builds each tree in a bottom-up manner (i.e., create leaf nodes at first and root node at last). Meanwhile, when building the k-tree, it exploits the information generated in building $(k+1)$-tree.

Algorithm 1 outlines `BottomUp`. We first initialize an empty forest $\mathcal{F}$, two arrays $pre[\,]$, $cur[\,]$, where $pre[v]$ and $cur[v]$ are supposed to keep the maximum value of l such that there is a (k, l)-core containing v. We also initialize the CUF

[1] In this paper, we use "node" to mean the "index tree node".

Algorithm 1 Index construction algorithm: `BottomUp`.

```
1: function BUILD(G)
2:     F ← ∅, pre[ ] ← ∅, cur[ ] ← ∅, Ψ ← ∅;
3:     for k ← k_max to 0 do
4:         map ← ∅, P ← ∅;
5:         cur[ ] ← DECOMPOSE(G, k);
6:         V_0, ··· , V_{l_max} ← group vertices of cur[ ];
7:         for l ← l_max to 0 do
8:             BUILDALEVEL(k, l, V_l, pre[ ], cur[ ], map, P, Ψ);
9:         create root node t and link to nodes in P;
10:        pre[ ] ← cur[ ], F.add(⟨t, map⟩);
11:    return F;
```

data structure Ψ which will be introduced later (line 2). Then, we enumerate k from $k_{\max}$ to 0 and for each k, we compute all the (k, l)-cores (lines 3-5), by using the algorithm in [Fang *et al.*, 2019b]. We initialize the vertex-node map map, and a set $\mathcal{P}$ for keeping the generated nodes for each level of k-tree (line 4). After that, we group vertices into a list of sets, such that V_l contains vertices which are in the (k, l)-core but not in the (k, l–1)-core (line 6). Next, we build the k-tree by invoking BUILDALEVEL to create nodes in the l-th level of k-tree where l ranges from $l_{\max}$ to 0 (lines 7-8). Finally, we create the root node with $\mathcal{P}$, and update $pre[\,]$ and $\mathcal{F}$ (lines 9-10).

Overview of function BUILDALEVEL. Given nodes in the (l+1)-th level of k-tree, the function BUILDALEVEL creates nodes in the l-th level and links them to the nodes in the (l+1)-th level. Since each node corresponds to a (k, l)-$\widehat{core}$, a naive method to check the connectivity and create the node will take $O(m)$ time to re-explore the graph, i.e., executing steps 2 and 3 of `TopDown`. Consequently, using this naive method totally takes $O(l_{\max} \cdot m)$ to build the k-tree, which is the same as that of `TopDown`.

To improve the efficiency, we propose a novel data structure, called Core-based Union-Find (or CUF), which allows the three key steps of BUILDALEVEL to be done efficiently: (1) verifying the connectivity, (2) memorizing the connectivity, and (3) linking nodes. In the following sections, we first introduce the CUF data structure, and then present our CUF-based BUILDALEVEL, which allows the k-tree to be built in $O(\alpha(n) \cdot m)$ time, where $\alpha(n)$ is the inverse Ackermann function and $\alpha(n) < 5$ for any practical value of n.

CUF data structure. CUF is extended from classic Union-Find (UF) Forest[2], which can efficiently verify the graph connectivity and partition vertices into different connected components. In the classic UF, each vertex has 2 elements, i.e., $rank$ and $parent$, and the UF has 3 functions, i.e., MAKESET, FIND and UNION, where MAKESET makes preparation for each vertex, FIND returns the representative member of the component to which the vertex belongs, and UNION merges two disjoint components as one. By using the classic UF, given a (k, 0)-core, we can verify the connectivity and sequentially find all (k, l)-$\widehat{core}$'s by varying l from $l_{\max}$ to 0, and then build all the levels of the k-tree accordingly. However, classic UF may have two main limitations.

[2]https://en.wikipedia.org/wiki/Disjoint-set_data_structure

Algorithm 2 Functions of the CUF data structure.

```
1: function MAKESET(v)
2:     v.rank ← 0, v.parent ← v;
3:     v.hook ← v, v.group ← v;
4: function FIND(v)
5:     if v.parent = v then v.parent ← FIND(v.parent);
6:     return v.parent;
7: function UNION(u, v, cur[ ])
8:     r_u ← FIND(u), r_v ← FIND(v);
9:     if r_u ≠ r_v then
10:        if r_u.rank < r_v.rank then SWAP(r_u, r_v);
11:        r_v.parent ← r_u;
12:        if r_u.rank = r_v.rank then r_u.rank ← r_u.rank + 1;
13:        if cur[r_u.group] < cur[r_v.group] then
14:            r_u.group ← r_v.group;
15: function UPDATECUF(V, cur[ ])
16:    for u ∈ V do
17:        r ← FIND(v);
18:        v.group ← r.group;
19:        if cur[r.hook] > cur[v] then r.hook ← v;
```

One limitation is that for a new node p in the l-th level, classic UF can not efficiently find p's all child nodes and link them up. As observed in Section 4.1, each subtree below the l-th level corresponds to a particular (k, l')-$\widehat{core}$ where $l' > l$. This means that to find p's child nodes, we can first find all (k, l')-$\widehat{core}$'s that are connected by vertices in $p.vSet$ and then link the root nodes of corresponding subtrees to p. To efficiently locate these subtrees, we assign another element $hook$ to directly indicate these root nodes. For example, for a vertex v in $p.vSet$, if v's neighbour u is contained in a (k, l')-$\widehat{core}$, we can locate the root node of the corresponding subtree by referring $hook$ and then link it to p.

The other limitation is that once the l-th level of the (k+1)-tree is constructed, the connectivity of the corresponding (k+1, l)-core is verified. When building the l-th level of the k-tree, we have to traverse the corresponding (k, l)-core and verify its connectivity from scratch. However, from Lemma 1, (k+1, l)-core is a subgraph of (k, l)-core, which implies that the connectivity of this (k+1, l)-core will be verified again. To cut off this redundant computation, we assign an additional element $group$ in CUF structure to "memorize" the particular (k, l)-$\widehat{core}$ to which each vertex used to belong. For instance, if vertex v is included in a certain (k+1, l)-$\widehat{core}$, $v.group$ will be marked; and when processing the (k, l)-core, by checking $v.group$, we would quickly know that v should be gathered together with others who share the same value.

Algorithm 2 summarizes all CUF functions and underlines our contribution. To maintain the two additional elements, we propose a new function UPDATECUF. As shown in Algorithm 2, for each vertex v, we find the representative member r of the particular (k, l)-$\widehat{core}$ including v, i.e., FIND(v) (lines 16-17). Then we update $v.group$ as $r.group$ and set $r.hook$ as v if v has smaller value in $cur[\,]$ (lines 18-19).

Lemma 5 (Space cost)**.** *Given a directed graph G, the CUF data structure of all vertices costs $O(n)$ space.*

Proof. The lemma directly follows the observation. □

Algorithm 3 Process vertices to create tree nodes.

```
1: function BUILDALEVEL(k, l, V_l, pre[ ], cur[ ], map, P, Ψ)
2:     initialize S_{v_1}, ···, S_{v_i}, ··· for vertices v_1, ···, v_i, ··· ∈ V_l;
3:     for v ∈ V_l do
4:         for u ∈ N(v) do
5:             if cur[u] > cur[v] then
6:                 r_u ← Ψ.FIND(u);
7:                 p' ← map.get(r_u.hook);
8:                 S_v.add(p'), P.delete(p');
9:     V' ← ∅;
10:    for v ∈ V_l do
11:        if k ≠ k_max and pre[v] = l then
12:            v.parent ← v, v.hook ← v;
13:            v.rank ← 0, V'.add(v);
14:        else Ψ.MAKESET(v);
15:    BATCHUNION(V_l\V', cur[ ], Ψ);
16:    for v ∈ V' do Ψ.UNION(v, v.group, cur[ ]);
17:    for each set C_i ⊆ V_l with the same CUF root do
18:        p ← create tree node by using l and C_i;
19:        P.add(p);
20:        for each v ∈ C_i do
21:            map.put(v, p);
22:            link nodes in S_v with p;
23:        Ψ.UPDATECUF(C_i, cur[ ]);
24: function BATCHUNION(V, cur[ ], Ψ)
25:    for v ∈ V do
26:        for u ∈ N(v) do
27:            if cur[u] ≥ cur[v] then Ψ.UNION(u, v, cur[ ]);
```

Lemma 6 (Time cost of CUF functions)**.** MAKESET *takes* $O(1)$ *time; for* UNION *and* FIND*, the amortized time per operation is* $O(\alpha(n))$*;* UPDATECUF *takes* $O(\alpha(n) \cdot |V|)$ *time.*

Proof. Obviously, MAKESET for each vertex takes $O(1)$. As for UNION and FIND, since they use *union by rank* and *path compression* optimization, the amortized time per operation is $O(\alpha(n))$ [Tarjan, 1979]. Thus, UPDATECUF totally takes $O(\alpha(n) \cdot |V|)$ for all vertices in V. □

Details of function BUILDALEVEL. Algorithm 3 shows the details. Firstly, we find root nodes of subtrees to be linked to new nodes in this level. For each vertex v, we initialize a set S_v; we visit v's neighbour u to find the root p' of the subtree including u and collect it in S_v (lines 2-8). Then we use CUF to verify the subgraph connectivity for this level. We initialize a set V' to collect vertices if they has been previously processed in the l-th level of the (k+1)-tree (lines 9-13). For vertices in V', we directly use their $group$ to achieve a quick UNION; for others, we visit their neighbours to check the connectivity by invoking BATCHUNION (lines 15-16, 24-27). After that, for each vertex set C_i sharing the same CUF root, we create the node p and update $\mathcal{P}$ (lines 17-19). For each vertex v in C_i, we update the vertex-node map map and link child nodes in S_v to p (lines 20-22). Finally, we update CUF for the constrcution of next level (line 23).

Lemma 7. *Given a directed graph G, the time cost of building D-Forest using* `BottomUp` *is* $O(\alpha(n) \cdot m \cdot \sqrt{m})$.

Proof. Decomposing the (k, 0)-core for a specific k takes $O(m)$ time [Fang *et al.*, 2019b]. Then BUILDALEVEL takes $O(\alpha(n) \cdot m_l)$, where m_l is the edges visited in the l-th level. This implies that building k-tree takes $O(\alpha(n) \cdot m)$ in total. Thus `BottomUp` takes $O(k_{\max} \cdot \alpha(n) \cdot m)$, which is upper bounded by $O(\alpha(n) \cdot m \cdot \sqrt{m})$. Lemma 7 holds. □

5 Experiments

5.1 Setup

We use six real large directed graphs in Table 1, where d is the average degree of vertices. The Twitter dataset [3] is collected by Kristina Lerman. The eu-2015, arabic, it-2004, sk-2005 and uk-2007 datasets [Boldi *et al.*, 2014] are available in the website[4]. We implement all the algorithms in Java, and run the experiments on a Linux machine with Ten-core Intel E7-4820 V3 CPU@1.90GHz and 300GB memory.

Graph	n	m	d	$k_{\max}$	$l_{\max}$
Twitter	699,986	36,743,091	52.49	443	448
eu-2015	6,650,532	165,693,531	24.91	9,568	9,569
arabic	22,744,080	639,999,458	28.14	3,126	3,126
it-2004	41,291,594	1,150,725,436	27.86	3,198	3,197
sk-2005	50,636,154	1,949,412,601	38.50	4,502	4,502
uk-2007	110,123,614	3,944,932,566	35.82	10,027	10,027

Table 1: Datasets in our experiments.

5.2 Experimental Results

We compare our approach with the state-of-the-art solutions [Fang *et al.*, 2019b] which propose three indices, i.e., `NestIDX`, `PathIDX` and `UnionIDX`, and three corresponding query algorithms, i.e., `Nest-Q`, `Path-Q` and `Union-Q`. Using the same evaluation strategy in [Fang *et al.*, 2019b], we respectively report the efficiency results of index construction and queries in Figures 3 and 4. To evaluate the efficiency of index construction, for each dataset, we randomly select 20%, 40%, 60% and 80% of its vertices and obtain the four subgraphs induced by these vertices.

Space cost of D-Forest. To measure the space cost of an index, we store all the index elements, which can be used to recover the index, into the disk. As shown in Figure 3 (a)-(f), with the sizes of sub-datasets growing, the space cost of D-Forest and others increase steadily. Besides, D-Forest uses comparable space cost as other indexes, meaning that D-Forest is as space-efficient as the state-of-the-art solutions.

Time cost of index construction. From Figure 3(g)-(l), we can observe that `BottomUp` takes similar time cost with state-of-the-art solutions . For instance, on the largest dataset uk-2007 (35.4GB in disk), `BottomUp` takes only 8.12 hours to build D-Forest. And `BottomUp` always runs at least 10 times faster than `TopDown`. We remark that to save the computational resources, for each test, we terminate `TopDown` if it runs 10 times longer than `BottomUp`.

Scalability evaluation of CSD queries. In Figure 4(a)-(f), we evaluate the scalability of CSD query algorithms over different sizes of datasets. As suggested in [Fang *et al.*, 2019b], for each sub-dataset, we randomly select 200 vertices which are in the (8,8)-cores as the query vertices and set k=l=8.

[3] https://www.isi.edu/lerman/downloads/twitter/twitter2010.html
[4] http://law.di.unimi.it/datasets.php

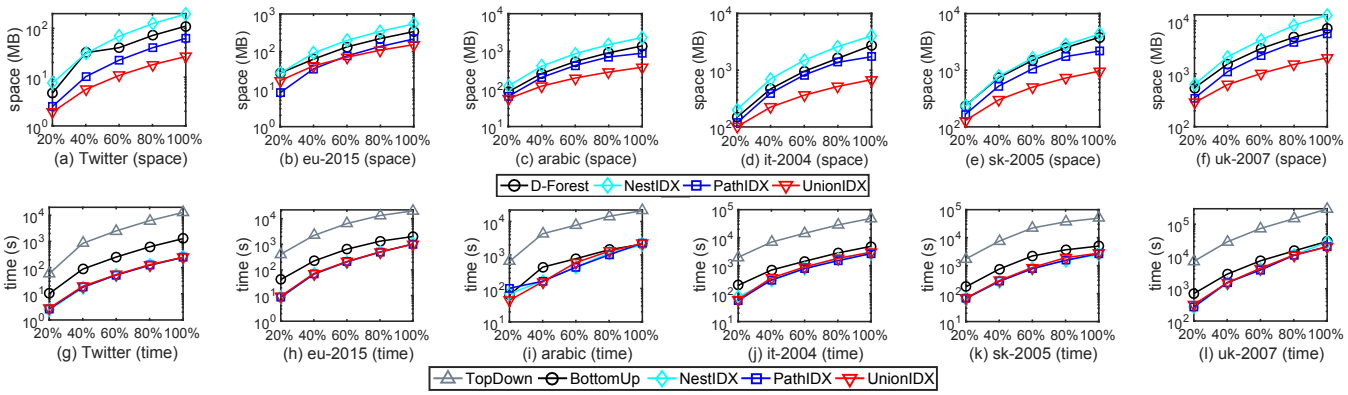

Figure 3: The space cost of D-Forest and the time cost of index construction.

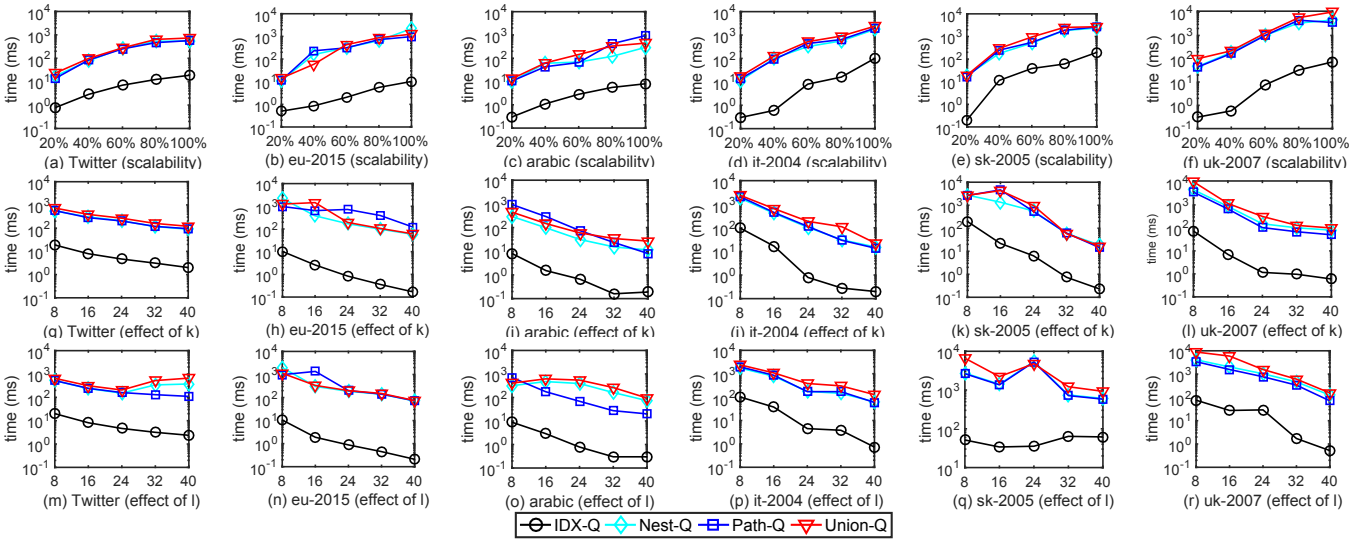

Figure 4: The efficiency of answering CSD queries.

Generally, the running time of all algorithms increases as the size of sub-datasets grows. Besides, our query algorithm `IDX-Q` achieves the best performance on all the datasets. For example, on uk-2007 dataset with around 4 billion edges, `IDX-Q` takes 72ms on average to answer a query and runs about 100 times faster than the existing solutions.

Effect of k and l in CSD queries. We depict the effect of k and l on the efficiency of CSD queries in Figure 4(g)-(r). For each dataset, we randomly select 200 vertices within the (8,8)-cores to query. We see that as the value of k and l increases, the returned communities become smaller, so the time cost of all query algorithms decreases. Again, `IDX-Q` is up to two orders of magnitude faster than the three existing algorithms which generally achieve similar performance.

6 Conclusion

In this paper, we examine the CSD problem and design a novel index D-Forest, based on which a CSD query can be completed in the optimal time cost. To build the index, we propose a basic algorithm `TopDown`, and an advanced algorithm `BottomUp` by introducing the CUF data structure. The experimental results on six real large graphs demonstrate the efficiency of our solutions.

In the future, we will investigate how to extend other cohesiveness metrics, such as k-truss and k-clique, to search communities in large directed graphs.

Acknowledgments

This work was conducted in CUHK supported by the Research Grants Council of the Hong Kong Special Administrative Region, China (CUHK 3133238 (Research Sustainability of Major RGC Funding Schemes)) and Delta-NTU Corporate Lab for Cyber-Physical Systems supported from Delta Electronics Inc and the National Research Foundation Singapore under the Corp Lab@University Scheme. Xin Cao was supported by ARC DE190100663. We would like to thank Jiani Zhang for her helpful discussion and proof-reading.

References

[Andersen and Lang, 2006] R. Andersen and K. Lang. Communities from seed sets. In *WWW*, pages 223–232. ACM, 2006.

[Batagelj and Zaversnik, 2003] V. Batagelj and M. Zaversnik. An o (m) algorithm for cores decomposition of networks. *CoRR*, 2003.

[Boldi *et al.*, 2014] P. Boldi, A. Marino, M. Santini, and S. Vigna. Bubing: massive crawling for the masses. In *WWW*, pages 227–228, 2014.

[Chen *et al.*, 2019] Y. Chen, Y. Fang, R. Cheng, Y. Li, X. Chen, and J. Zhang. Exploring communities in large profiled graphs. *TKDE*, 31(8):1624–1629, 2019.

[Cui *et al.*, 2013] W. Cui, Y. Xiao, H. Wang, Y. Lu, and W. Wang. Online search of overlapping communities. In *SIGMOD*, pages 277–288, 2013.

[Cui *et al.*, 2014] W. Cui, Y. Xiao, H. Wang, and W. Wang. Local search of communities in large graphs. In *SIGMOD*, pages 991–1002, 2014.

[Ebadian and Huang, 2019] S. Ebadian and X. Huang. Fast algorithm for k-truss discovery on public-private graphs. In *IJCAI*, pages 2258–2264, 2019.

[Fang *et al.*, 2017a] Y. Fang, R. Cheng, Y. Chen, S. Luo, and J. Hu. Effective and efficient attributed community search. *VLDB Journal*, 26(6):803–828, 2017.

[Fang *et al.*, 2017b] Y. Fang, R. Cheng, X. Li, S. Luo, and J. Hu. Effective community search over large spatial graphs. *PVLDB*, 10(6):709–720, 2017.

[Fang *et al.*, 2017c] Yixiang Fang, Reynold Cheng, Siqiang Luo, Jiafeng Hu, and Kai Huang. C-explorer: browsing communities in large graphs. *PVLDB*, 10(12):1885–1888, 2017.

[Fang *et al.*, 2019a] Y. Fang, X. Huang, L. Qin, Y. Zhang, W. Zhang, R. Cheng, and X. Lin. A survey of community search over big graphs. *VLDB Journal*, 2019.

[Fang *et al.*, 2019b] Y. Fang, Z. Wang, R. Cheng, H. Wang, and J. Hu. Effective and efficient community search over large directed graphs. *TKDE*, 31(11):2093–2107, 2019.

[Fang *et al.*, 2019c] Yixiang Fang, Zheng Wang, Reynold Cheng, Xiaodong Li, Siqiang Luo, Jiafeng Hu, and Xiaojun Chen. On spatial-aware community search. *TKDE*, 31(4):783–798, 2019.

[Fang *et al.*, 2019d] Yixiang Fang, Kaiqiang Yu, Reynold Cheng, Laks V.S. Lakshmanan, and Xuemin Lin. Efficient algorithms for densest subgraph discovery. *PVLDB*, 12(11):1719–1732, 2019.

[Fang *et al.*, 2020] Yixiang Fang, Yixing Yang, Wenjie Zhang, Xuemin Lin, and Xin Cao. Effective and efficient community search over large heterogeneous information networks. *PVLDB*, 13(6):854–857, 2020.

[Flake *et al.*, 2000] G. W. Flake, S. Lawrence, and C. L. Giles. Efficient identification of web communities. In *SIGKDD*, volume 2000, pages 150–160, 2000.

[Fortunato, 2010] S. Fortunato. Community detection in graphs. *Physics reports*, 486(3-5):75–174, 2010.

[Giatsidis *et al.*, 2011] C. Giatsidis, D. M. Thilikos, and M. Vazirgiannis. D-cores: Measuring collaboration of directed graphs based on degeneracy. In *ICDM*, pages 201–210, 2011.

[Hu *et al.*, 2017] Jiafeng Hu, Xiaowei Wu, Reynold Cheng, Siqiang Luo, and Yixiang Fang. On minimal steiner maximum-connected subgraph queries. *TKDE*, 29(11):2455–2469, 2017.

[Hu *et al.*, 2019] Jiafeng Hu, Reynold Cheng, Kevin Chen-Chuan Chang, Aravind Sankar, Yixiang Fang, and Brian YH Lam. Discovering maximal motif cliques in large heterogeneous information networks. In *ICDE*, pages 746–757. IEEE, 2019.

[Huang and Lakshmanan, 2017] X. Huang and L. Lakshmanan. Attribute driven community search. *PVLDB*, 10(9):949–960, 2017.

[Huang *et al.*, 2014] X. Huang, H. Cheng, L. Qin, W. Tian, and J. Yu. Querying k-truss community in large and dynamic graphs. In *SIGMOD*, pages 1311–1322, 2014.

[Huang *et al.*, 2015] X. Huang, L. Lakshmanan, J. X. Yu, and H. Cheng. Approximate closest community search in networks. *PVLDB*, 9(4):276–287, 2015.

[Kim *et al.*, 2010] Y. Kim, S. Son, and H. Jeong. Finding communities in directed networks. *Phys. Rev. E*, 81(1):016103, 2010.

[Lancichinetti and Fortunato, 2009] A. Lancichinetti and S. Fortunato. Benchmarks for testing community detection algorithms on directed and weighted graphs with overlapping communities. *Phys. Rev. E*, 80(1):016118, 2009.

[Leicht and Newman, 2008] E. A Leicht and M. E. Newman. Community structure in directed networks. *Phys. Rev. letters*, 100(11):118703, 2008.

[Li *et al.*, 2015a] R. Li, L. Qin, J. Yu, and R. Mao. Influential community search in large networks. *PVLDB*, 8(5):509–520, 2015.

[Li *et al.*, 2015b] Zhenguo Li, Yixiang Fang, Liu Qin, Jiefeng Cheng, Reynold Cheng, and John C.S. Lui. Walking in the cloud: parallel simrank at scale. *PVLDB*, 9(1):24–35, 2015.

[Malliaros and Vazirgiannis, 2013] F. Malliaros and M. Vazirgiannis. Clustering and community detection in directed networks: A survey. *Physics Reports*, 533(4):95–142, 2013.

[Newman and Girvan, 2004] M.E. Newman and M. Girvan. Finding and evaluating community structure in networks. *Phys. Rev. E*, 69(2):026113, 2004.

[Ning *et al.*, 2016] X. Ning, Z. Liu, and S. Zhang. Local community extraction in directed networks. *Phys. A: Statist. Mech. Appl.*, 452:258–265, 2016.

[Sozio and Gionis, 2010] M. Sozio and A. Gionis. The community-search problem and how to plan a successful cocktail party. In *SIGKDD*, pages 939–948, 2010.

[Tarjan, 1979] R. E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of computer and system sciences*, 18(2):110–127, 1979.

[Wan *et al.*, 2020] Guojia Wan, Bo Du, Shirui Pan, and Jia Wu. Adaptive knowledge subgraph ensemble for robust and trustworthy knowledge graph completion. *WWW*, 23(1):471–490, 2020.

[Wang and Zhu, 2019] C. Wang and J. Zhu. Forbidden nodes aware community search. In *AAAI*, pages 758–765, 2019.

[Yang *et al.*, 2010] T. Yang, Y. Chi, S. Zhu, Y. Gong, and R. Jin. Directed network community detection: A popularity and productivity link model. In *SIAM*, pages 742–753, 2010.

[Yang *et al.*, 2014] J. Yang, J. McAuley, and J. Leskovec. Detecting cohesive and 2-mode communities indirected and undirected networks. In *WSDM*, pages 323–332. ACM, 2014.

[Zhang *et al.*, 2014] J. Zhang, C. Wang, and J. Wang. Who proposed the relationship?: recovering the hidden directions of undirected social networks. In *WWW*, pages 807–818, 2014.

[Zhu *et al.*, 2019] Qikui Zhu, Bo Du, and Pingkun Yan. Multi-hop convolutions on weighted graphs. *arXiv preprint arXiv:1911.04978*, 2019.