IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received August 13, 2021, accepted August 29, 2021, date of publication September 3, 2021, date of current version September 14, 2021.

*Digital Object Identifier 10.1109/ACCESS.2021.3109862*

# Influential Attributed Communities Search in Large Networks (InfACom)

**NARIMAN ADEL HUSSEIN, HODA M. O. MOKHTAR, AND MOHAMED E. EL-SHARKAWI**
Information Systems Department, Faculty of Computers and Artificial Intelligence, Cairo University, Cairo 12613, Egypt
Corresponding author: Nariman Adel Hussein (n.adel@pg.cu.edu.eg)

**ABSTRACT** Community search is a fundamental problem in graph analysis. In many applications, network nodes have specific properties that are essential for making sense of communities. In these networks, attributes are associated with nodes to capture their properties. The community influence is a key property of the community that can be employed to sort the communities in a network based on the relevance/importance of certain attributes. Unfortunately, most of the previously introduced community search algorithms over attributed networks neglected the community influence. In this paper, we study the influential attributed community search problem. Different factors for measuring the influence are discussed. Also, different Influential Attributed Community (InfACom) algorithms based on the concept of k-clique are proposed. Two techniques are presented one for sequential implementation with three variations and one for parallel implementation. In addition, we propose efficient algorithms for maintaining the proposed algorithms on dynamic graphs. The proposed algorithms are evaluated on different real datasets. The experimental results show that the summarization technique reduces the size of the graph by approximately half. In addition, it shows that the proposed algorithms *EnhancedExact* and *Approximate* outperform the state-of-the-art approaches Incremental Time efficient ($Inc - T$), Incremental Space efficient ($Inc - S$), *Exact*, and 2-Approximation (*AppInc*) in both efficiency and effectiveness. For the *EnhancedExact* algorithm, the results show that the efficiency is at least 7 times faster than the $Inc - S$ algorithm, at least 4.5 times faster than the $Inc - T$ algorithm, and 2 times faster than the *AppInc* algorithm. For the *Approximate* algorithm, the results show that its efficiency is at least 10 times faster than the $Inc - S$ algorithm, at least 6.4 times faster than the $Inc - T$ algorithm, and 3 times faster than the *AppInc* algorithm. Finally, the results show that the proposed algorithms retrieve cohesive communities with a smaller diameter than all the state-of-the-art approaches.

**INDEX TERMS** Community search, k-clique percolation community, summary graph, node attribute, keyword search.

## I. INTRODUCTION

In recent years, due to the excessive use of numerous real networks such as social networks, a large amount of complex data has been generated. For example, Facebook Friendship Network has more than 2.74 billion monthly active users worldwide [16]. As of the first quarter of 2019, Twitter has more than 330 million monthly active users [12]. Due to this extraordinary rise in the volume of data, it is crucial to design a method to efficiently detect hidden patterns among the group of users. Graph data models played a critical role in big data analysis in recent years [5]. In real-world applications, graphs are used to represent different types of entities. These entities generally have attributes that are important for understanding the community using graphs. For example, a node on an academic collaboration network like DBLP indicates an author and the edge between the two authors indicates a collaboration relationship. Nodes can also have attributes that represent their area of expertise. These networks can be modeled using attributed graphs [48] where the attributes associated with the node capture its properties.

A community can be defined as a set of vertices (nodes) which probably share common features, where the nodes in the same communities have a dense connection with each other than that of different communities [20], [24]. The task of finding communities can be divided into two major classes: community detection, and community search. Community detection methods usually use global criteria to detect all the

The associate editor coordinating the review of this manuscript and approving it for publication was Mehdi Hosseinzadeh.

communities from an entire graph, where the focus is more on quality (e.g., cohesiveness) than efficiency. [7], [19], [20], [28], [40], [41], [44], [45]. Community search is a fundamental problem in graph analysis [2]. Community search is a query-dependent variant from the community detection problem. In community search, communities are defined based on query conditions, and community search solutions aim to find communities efficiently in an online manner [42]. Different types of cohesive subgraphs are used as building blocks for communities in graph such as k-core [10], [17], [18], [31], [39], [46], k-truss [1], [23], [25]–[27], [36], [47], and clique or quasi-clique [13], [43].

In all previous studies on community detection and community search, a community is defined as a densely connected subgraph. This ignores another important aspect, namely the ''influence'' of a community. Community influence can be defined as a key community property that can be used to classify communities in the network based on the relevance/importance of certain attributes. Recently, detecting influential communities was studied. It was first introduced by Li *et al.* in [34] later it was investigated in [3], [6], [8], [32], [35]. The common goal of finding influential communities is to find a closely connected group of users (vertices) who have some dominance over other users in the graph in a particular domain. Previous attempts have defined the influence of the community as the minimum weight of its nodes, where the weight denotes the influence (importance) of the node. Traditional community search research that considered attributed graph have the following drawbacks:

1) Require an input query vertex, and then find a group of neighboring vertices whose attributes have high similarity with the query vertex attributes. A major limitation of such community search techniques is that the user needs to define the query vertex which might not be possible or suitable in many application domains.
2) Another type of community search solution finding cohesive communities having close similarity with query attributes. However, they do not consider the influence of the community.
3) The other type of community search solutions only work on non-attributed graphs and considers the influence of the community as the minimum weight of its nodes, where the weight denotes the influence (importance) of the node. However, this assumption ignores the relationship between nodes. Also, it fails to express the actual influence of the nodes in a community with respect to its associated attributes.
4) Effective storage utilization is another challenge that faces the problem of community detection/search. Existing models require maintaining the original graph and/or its index. Yet such information is considerably huge to keep in memory.

To fill the above research drawbacks, we propose a new community search approach called Top-r Influential Attributed communities (InfACom) in large networks. The main contributions of the paper are the following:

1) Proposing a new influence measure for a community that considers the attribute weight (the influence value of the node in this attribute), the attribute connectivity (the degree of connectivity between nodes of the same attribute), and the relationships between nodes.
2) Proposing a new approach for graph summarization to improve memory usage.
3) Proposing different community search algorithms that retrieve the influential communities.
4) Proposing efficient algorithms for managing dynamic networks.
5) Designing a parallel approach that decomposes the original graph into different subgraphs and allows searching for the influencing communities in parallel on these subgraphs.
6) Conducting experiments on real datasets to evaluate the proposed algorithms. The experimental results show that the proposed algorithms are highly efficient and effective in retrieving influential communities compared to the state-of-art approaches.

The rest of this paper is organized as follows: Section II presents the needed preliminaries and some related definitions. Section III presents related work. Section IV discusses the influence calculation in attributed community. In Section V, we discuss different influential attributed community techniques. In Section VI, we explain the summary graph update algorithms. Our experimental results are presented in Section VII. Finally, Section IX gives a summary, critique of the findings, and proposes directions for future work.

## II. PRELIMINARIES

In this section, we present preliminaries and some related definitions that will be used in the rest of this paper. Section II-A presents some graph concepts. Graph summarization concept is discussed in Section II-B.

### A. BASIC GRAPH CONCEPTS

We consider an undirected, unweighted graph $G = (V, E)$, where $V$ represents the set of nodes (also called vertices) and $E$ represents the set of edges in $G$. We denote the number of nodes and number of edges of graph $G$ by $n$ and $m$ respectively, i.e., $n = |V|$ and $m = |E|$. For each node $u \in V$, we use $N(u)$ to denote the set of neighbors of $u$ in $G$, i.e., $N(u) = v|(u, v) \in E$. The degree of a node $u \in V$, denoted by $d(u)$, is the number of neighbors of $u$ in $G$, i.e., $d(u) = |N(u)|$. Given a set of node $V_s$ in $G$, the node-induced subgraph by $V_s$, denoted by $G_{Vs} = (V_s, E_s)$, is a subgraph of $G$, such that $G_{Vs} = (V_s, (u, v) \in E|u, v \in V_s)$. A graph G is complete if and only if its vertices are pairwise adjacent, i.e. $\forall v_i, v_j \in V,\ (v_i, v_j) \in E$ and $i \neq j$. The density of the complete graph is 1.

*Definition 2.1 (*`Graph Density`*):* The density of the graph is a measurement of the number of connections between nodes compared to the number of possible connections between the nodes. [13]. The density of an undirected

graph is calculated as,

$$Density = \frac{2m}{n(n-1)} \quad (1)$$

In this paper, we aim to find the influential communities in an attributed graph.

*Definition 2.2 (*`Attributed Graph`*):* An attributed graph is a 3-tuple $G = (V, A, E)$, where $V$ is the set of vertices, $E$ is the set of edges such that $e_{ij}$ is an edge between nodes $v_i$, $v_j$, and $A = a_1, \ldots, a_n$ is a set of attributes associated with $V$. Such that for each $u \in V$, there is an attribute vector $A(u) = (a_1(u), \ldots, a_n(u))$ is associated with $u$, where $a_l(u)$ is the attribute value of $u$ on the *ith* attribute $a_i (1 \leq i \leq n)$.

In practice, the graphs are continuously evolving. Thus attributes and edges of graphs are frequently updated.

*Definition 2.3 (*`Dynamic Graph`*):* Is a graph that is subject to a series of updates such as inserting or deleting graph edges or vertices, or changing attributes associated with nodes.

*Definition 2.4 (*`A Community`*):* Is a set of vertices (nodes) which probably share common features, where the nodes in the same community have dense connections with each other than that of different communities [24], [30].

Different types of cohesive subgraphs are used as building blocks for communities in graph such as k-core [17], [18], [46], k-truss [1], [23], [25], [26], [36], [47], and clique or quasi-clique [13], [43], In this paper, we consider the k-clique community as it has the most cohesive structure. A k-clique is a complete subgraph of k nodes, which has a minimum diameter of 1 and a maximum density of 1. Furthermore, k-clique has the following properties: (1) each node has at least (k-1) neighbors; (2) After removing edges less than (k-2), the graph remains connected; (3) Each pair of nodes has at least (k -2) neighbors. Therefore, the k-clique is a subgraph of (k- 1)-core, and (k- 2)-truss, indicating that it has the strongest cohesiveness.

*Definition 2.5 (*`k-Clique`*):* A k-clique is a complete subgraph of k vertices, where each pair of vertices is adjacent. A k-clique is the densest graph among all k-node graphs [13], [38].

In many applications, communities can overlap each other, such as studying cancer-related proteins in protein interaction networks [29], and the evaluation of stock correlations [21].

*Definition 2.6 (*`k-Clique Adjacency`*):* Given 2 cliques $C_i$ and $C_j$ in G, $C_i$ and $C_j$ are adjacent if they share (k-1) nodes, i.e., $|C_i \bigcap C_j| = (k-1)$ [13], [38].

*Definition 2.7 (*`k-Clique Adjacency Graph`*):* Given an undirected graph $G$ and an integer $k$, the k-clique adjacency graph of $G$ denoted by $G_C = (V_C, E_C)$ such that each node $v_i \in V_C$ is a k-clique in G and there is an edge $e(v_i, v_j)$ if the corresponding k-cliques $C_i$ and $C_j$ in $G$ are adjacent [13], [38].

*Definition 2.8 (*`k-Clique Component`*):* Given the k-clique adjacency graph $G_C$, the k-clique component is the series of k-cliques $C_1, \ldots, C_n$ such that $C_i$ and $C_{i+1}$ are adjacent for $1 \leq i \leq n$ [13], [38].

We consider the `k-clique components` as the influential communities, where the influence of the community $inf(H)$ is measured using Equation 4 (see Section IV). k is termed as cohesion factor.

*Definition 2.9 (*`k-Influential Community`*):* Let $G = (V, A, E)$ an undirected, attributed graph and an input k, the k-influential community is an induced subgraph H of G that satisfies the following:

- *Connectivity:* $H$ is a connected graph; there is a path between pair of vertices.
- *Structure Cohesiveness:* $H$ is a dense subgraph (k-clique component).
- *Maximal Structure:* H is a maximal induced subgraph, where there is no other induced subgraph $H'$ such that (1) $H'$ satisfies connectivity and cohesiveness constraints, (2) $H'$ contains $H$, and (3) $inf(H') = inf(H)$.

Now, we define the top-r k-Influential attributed communities (InfACom) as follows:

*Definition 2.10 (*`InfACom Search Problem`*):* Given an undirected, attributed graph $G = (V, A, E)$, and two parameters k and r, where k represents structure cohesiveness (i.e., its vertices are tightly connected) and r be a positive integer specifying the number of top communities to be returned. Then, InfACom finds the top-r influential communities $\{H_1, H_2, \ldots\ldots\ldots, H_r\}$ with highest influence score with respect to some attribute $a_i$ in A.

*Example 2.11:* Consider an attributed graph of researchers, as shown in Figure 1, where nodes represent authors who published papers in research fields related to "Artificial Intelligence (AI)", and "Database (DB)". A Ph.D. student may be interested in finding the influential community who are working in "DB". The community $[v_1, v_2, v_3, v_7, v_8]$ is returned as a top-2 influential community with k = 4 (see Section V for the influential calculations) since the members of the community have influence in "DB", the community is dense and also contains highly influential members.

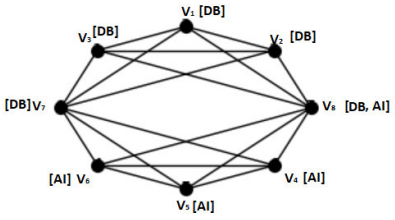



**FIGURE 1.** An attributed graph G.

### B. GRAPH SUMMARIZATION

At the current growth rate of data volume, it is very impractical to store, process, analyze, and visualize these large graphs. Therefore, in order to make graph data management, processing, and visualization tractable, summarization techniques are becoming increasingly important. Graph summarization techniques aim for reducing the size of the graph, then it

can be loaded into the memory to improve the performance of analytics algorithms. In addition, many graph algorithms that are complex or expensive to run on larger graphs can be efficiently executed on summary graphs. one of the key challenges of graph summarization is that it can have a serious impact on the amount of ''useful information'' represented by the graph. Several techniques [37] have been introduced to summarize graphs such as grouping-based, bit-based compression, simplification-based, and influence-based methods. Grouping-based methods aggregate nodes into supernodes connected by superedges according to the structural characteristics and attributes of the nodes. Grouped nodes are close to each other in structure and have similar attributes. Some methods use existing clustering techniques to find and assign clusters to supernodes. Others use an application-dependent optimization function to create the supernodes and superedges. Bit compression is a common technique for data mining. Such a method aims to minimize the number of bits required to describe a particular graph. Simplification-based methods simplify the original graph by deleting less important nodes or edges, which in turn produces a sparse graph. The summary graph consists of a subset of the original nodes and/or edges. Several algorithms can be based on simplified summaries, such as sampling and sketching. Finally, influence-based summarization methods for labeled graphs are currently scarce. These methods leverage both structural and node attribute similarities to summarize the influence or diffusion process in large networks.

## III. RELATED WORK

This paper classifies the related work into the following three main categories: community search, attributed community search, and influential community search.

### A. COMMUNITY SEARCH

Community search aims to find the community that contains a specific query node. Mining cohesive subgraphs is one of the most fundamental graph problems which aims to find groups of well-connected nodes. A variety of models have been proposed such as quasi-cliques, k-core, and k-truss [2], [14], [23], [26]. All these approaches focus only on the community structure while ignoring node attributes as well as community influence. In [2], an indexing approach is proposed for solving the Minimum-Core community search. First, an index is constructed to hold the structural information of all k-cores and then develop a heuristic strategy to connect all query nodes into a candidate community and refine it. This approach contains two main steps: preprocessing and query processing. In the preprocessing step, a Shell-Index is constructed to precompute and store some useful information for query processing. In the query processing step, the proper information is retrieved from the index and processed to obtain the answer to the query. Cui *et al.* [14] proposed a local-search algorithm to improve the efficiency of the global-core algorithm. This algorithm iteratively expands the neighbors of a single query vertex, until a subgraph containing the optimal solution is constructed. Then, this subgraph is used as a reduced version of the input graph to find the best solution. In [23], a novel Triangle Connectivity Preserving (TCP) Index was designed to find all overlapping communities of a given query vertex. However, they ignore the diameter of the resulting community. Finally, the problem of closest community search was studied in [26] based on k-truss and graph diameter (measure closeness of the nodes). The proposed community model requires that all query nodes are connected in this community.

### B. ATTRIBUTED COMMUNITY SEARCH

Community search over attributed graphs problem has attracted much attention in recent years [9], [11], [17], [18], [25], [46], [49]. The community search problem for a profiled graph was investigated in [9]. The profiled graph is an attributed graph in which each vertex is associated with a set of labels arranged hierarchically called a P-tree. A Core Profiled tree (CP-tree) index was constructed by considering both the graph vertices and P-trees of a profiled graph. Each CP-tree node corresponds to a label and stores the k-cores sharing this label. The keyword search-based method was proposed in [11] to generate a community with a structure and attribute cohesiveness. First, the influential attributes are derived according to the probability of occurrence of each pair of attributes type-value to all nodes and edges. The Attribute Index Structure was created according to all the attribute information on nodes. The attributes are classified into separate groups based on the attribute similarity between the nodes, and their degree structure. Based on this index a method for determining the community of nodes in an online manner was created. In [17] the problem of finding a spatial-aware community (SAC) was studied. SAC is a community with high structure cohesiveness and spatial cohesiveness. Given a query vertex q, the goal is to find a SAC containing q in the smallest minimum covering circle MCC (smallest radius) and all vertices of SAC satisfy the minimum degree metric which is a metric used to measure the structure cohesiveness. The authors proposed exact solutions for finding a SAC that contains q as well as approximation algorithms to compute a SAC for large datasets. In [18], the attributed community query (ACQ) problem was investigated, which returns an attributed community (AC) in an attributed graph. A tree-based index was developed to enable efficient AC search. The index is a compressed tree that is built based on the key observation whose cores are nested. The nodes of the tree are further augmented by inverted lists. This index was created using two approaches (top-down, and bottom-up). The problem of finding connected and close k-truss subgraphs containing query nodes was investigated in [25] over attributed graphs, with the largest attribute relevance score. First, an attribute score function satisfying the desirable properties of a good attributed community was introduced. Then, an efficient greedy algorithmic framework was developed to find the community containing given query nodes according to the given query attributes based on the maximal

(k, d)-truss. Finally, an index was built to maintain k-truss structure and attribute information to efficiently answer the given query. The problem of finding (k, r)-core was investigated in [46], which intends to find cohesive subgraphs on social networks considering both user engagement and similarity perspectives. Several novel pruning techniques were proposed to enumerate the maximal (k, r) efficiently. Finally, the problem of geo-social group search based on spatial containment was investigated in [49]. Given a set of spatial query points and a social network, the query is to find a minimum group of users whose members satisfy a certain social relationship and whose associated regions can jointly cover all the query points. A novel Social-aware R-tree (SaR-tree) was designed where each entry maintains some aggregate social relation information for the users covered by this entry.

### C. INFLUENTIAL COMMUNITY SEARCH

Influential community search aims to discover a closely connected group of nodes (vertices) that have some dominance over other nodes in the graph in a specific domain [3], [33], [34]. In [3] an instance-optimal algorithm was proposed to compute the top-k influential communities without using indexes. In [32], a new community model was proposed that reveals the communities with the highest outer influences. Also, a tree-based index structure and different algorithms were developed to improve search performance. In [33], a new skyline community model is proposed that detects communities in a multi-valued network, where each node is associated with d numerical attributes. Finally, Li *et al.* [34] proposed a new community model based on the concept of k-core, called the k-Influential community. This model considered the importance of nodes. To find the k-Influential community, the key idea of the proposed solution is to build an index that combines the importance of nodes and the k-core structure. Also, an index-based online query processing algorithm was introduced to quickly identify the k-Influential community that contains query nodes.

## IV. INFLUENCE CALCULATION IN ATTRIBUTED COMMUNITY

As discussed in previous sections, the current attempts in graph influence calculations do not explain the reasoning behind associating an influence value. In this section, we discuss the main factors that affect the influence value of a community.

For the following discussion, let $G = (V, A, E)$ be an attributed graph, where $v_i$ is associated with a set of attributes denoted by $a_{v_i}$. In this paper, we consider attributes representing sets of interesting topics of nodes. For example, attribute-set {DB, AI} associated with node $v_i$ expresses that $v_i$ is interested in topics DB and AI.

A vertex weighted $G_w$ is an undirected graph such that for each vertex $v_i \in V$ there exists a vector of weights $[a_{1w_i} a_{2w_i} \ldots\ldots a_{nw_i}]$ such that $a_{j_{w_i}}$ is the weight of attribute $a_j \in A$ for a node $v_i$ such that $a_{j_{w_i}}$ is calculated using Eq.2.

$$a_{j_{w_i}} = \alpha + \frac{|V'|}{|N(v_i)|} + \gamma \frac{|V''|}{|V_{NN}|} \tag{2}$$

where,

$$\alpha = \begin{cases} 1 & a_j \in v_{iA} \\ 0 & \text{Otherwise} \end{cases}$$

$N(v_i)$ represents the set of direct neighbors of $v_i$. $V'$ represents the set of direct neighbors of $v_i$ that contains the attribute $a_j$, $V' = \{v' \in V : (v', v_i) \in E, a_j \in v'_A\}$,

$V_{NN}$ represents the set of the next neighbors of $v_i$, $V_{NN} = \{v_{NN} \in V : (v_{NN}, v_k), (v_k, v_i) \in E\}$

$V''$ represents the set of next neighbors of $v_i$ that contains the attribute $a_j$, $V'' = \{v'' \in V : (v'', v_k), (v_k, v_i) \in E | a_j \in v''_A\}$, and $\gamma$ is a tuning parameter. In the general case, the value of $\gamma$ can be tuned depending on the number of hops of interest. For the rest of the paper, we assume that $\gamma = 0.5$ to reflect two hops influence.

The influence value of a community is measured based on the following factors:

1) *The Average Weight of an Attribute:* Given a subgraph $G(V, A, E)$ and an attribute $a_j$, the average weight of this attribute in G is given by

$$avg(a_{j_G}) = \frac{\sum a_{j_{w_i}}}{|V|} \quad \forall i \in V \tag{3}$$

Consider graph G in Figure 2. Since node $v_1$ has the largest number of direct neighbors and next neighbors that contain the topic "DB" thus it has more influence on "DB" than other nodes in the graph.

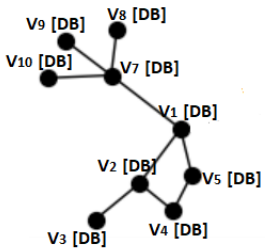



**FIGURE 2.** Graph G represents the role of neighbors.

2) *The Strength of the Relationship Between Nodes:* The authors in [26], [45] explained an undesirable phenomenon called the "free-rider effect": the detected community contains irrelevant nodes. To avoid this phenomenon, the diameter $Diam(C)$, an important feature of a community, is taken into consideration [15], [26] to measure the closeness of the nodes. The diameter of graph G is the shortest path between the farthest 2 nodes in G.
3) *The Effective Density of the Attributed Community:* Given the fact that a node can exist in more than one

community (also known as community overlap), computing the contribution of the attribute in an overlapping community is a crucial measure to determine the effective density of the community.

The density of a community represents the degree of connectivity among the community nodes. The density of a graph $G = (V, E)$ is measured by $\frac{2|E|}{|V|(|V|-1)}$. Having an attributed community, the effective density of the attributed community is a measure of the connectivity among the nodes having the same attribute among the attribute list.

To compute the effective density of attributed community, we first introduce the following definitions:

*Definition 4.1 (*`Effective Node`*):* An effective node $v_{eff}$ is the node that has a set of attributes $A'$ such that the query attribute $a_Q \subseteq A'$.

*Definition 4.2 (*`Effective Edge`*):* An effective edge $e_{eff}$ is the edge between 2 effective nodes.

Hence, the effective density

$$\sigma(G) = \frac{2|e_{eff}|}{|v|(|v|-1)}.$$

Given the above measures, the community influence of an attributed undirected graph G(V,A,E) is given as

$$inf(G) = \sigma(G)\left(\frac{avg(a_{j_G})}{diam(G)}\right) \tag{4}$$

## V. INFLUENTIAL ATTRIBUTED COMMUNITY SEARCH (InfACom)

This section introduces two different techniques for solving the problem presented in Definition 2.10: Whole InfACom, and Decomposed InfACom. Whole InfACom deals with the given graph as one part. Although this approach yields high accuracy, yet it is time-consuming. On the other hand, decomposed InfACom is a parallel technique that first decomposes the given graph into smaller partitions, and then processes them in parallel, aggregates results, and finally returns the final answer (communities).

### A. WHOLE InfACom

In this section, we present Whole InfACom a sequential technique that consists of two main modules: graph summarization module and community search module.

#### 1) GRAPH SUMMARIZATION MODULE

The graph summarization module generates a much compact and informative graph that summarizes the structural characteristics of the original graph. It summarizes the original graph by merging nodes and edges from the original graph into node groups (supernodes) and group relationships (superedges), respectively. Using the summary graph, all structure information, as well as the attribute information, can be restored.

*Definition 5.1 (*`The Summary Graph`*):* Given an undirected graph G = (V,A,E), the summary graph is denoted as $G_s = (S, E_s)$. Each node of the summary graph, called a supernode, is a complete component in G. Each supernode has the following metadata: a) MemberNodes (MNodes): is the set of nodes in the supernode containing the weights of the attributes, b) NeighboringNodes (NNodes): is a list that stores an entry for each node in MNodes along with its neighbors in G excluding the nodes existing in the MNodes, c) AttList: which is an inverted list that stores an entry for each attribute that the supernode has alongside the other nodes containing it.

The superedge is a weighted edge that represents a relationship between two supernodes. A superedge $e_{ij}$ exists between two supernodes $S_i$ and $S_j$ if and only if there exists at least one edge connecting any node in MNodes of $S_i$ with any node in MNodes of $S_j$.

There are two main steps for creating the summary graph:

1) *Create Supernodes:* This step aims to create supernodes of $G_s$ where each supernode is a complete component that summarizes a large number of nodes from G. First, the node with the maximum degree is determined. Then, the algorithm checks out the neighbors of this node for creating a complete graph. The neighbors are checked one by one according to their degree. The neighbors with the highest degree are checked first so that the algorithm can summarize the largest number of nodes together into one supernode. Once a complete component is created, a supernode is created with its metadata. Then, all nodes in MNodes are marked as `visited` in G. This operation is repeated until no unvisited nodes are left in G. While creating the supernodes, two supplementary data structures are constructed: 1) NodeList which is a list that stores an entry for each node in G along with the identifier of the supernode that contains it, and 2) MapList which is a list that stores an entry for each attribute in the graph along with the supernodes containing it as well as the maximum weight of the attribute in this supernode.
2) *Create Superedges:* This step creates a superedge by adding a weighted edge between any two supernodes $S_i$ and $S_j$, if there are nodes shared between the NNodes of $S_i$ and the MNodes of $S_j$. The weight represents the number of shared nodes.

Details are given in Algorithm 1, which is described below.

Algorithm 1 calls two main Procedures: `CreateNodes` that is responsible for creating supernodes and `Connect Nodes` that is responsible for creating superedges. `CreateNodes` first calls `FullComponent` that is responsible for generating a fully connected component (step 6). First, it finds the unvisited node `u` with the highest degree in the graph `G` and saves it in an empty set `S`. Then, it finds all its unvisited neighbors `UN`. Each node $v_i$ of `UN` is checked that it is connected to the nodes stored in S. If yes, $v_i$ is saved in S. Once the condition fails, `FullComponent` returns `MNodes` with the maximal full component that contains `u`. Then, `NNodes` and `AttList` are generated with the neighbors and attributes of all nodes in `MNodes`, respectively (steps 7-8). $S_i$ is created with its metadata and

the supplementary lists (NodeList and MapList) are updated. Also, $S_i$ is added to $G_s$ (steps 9-11). Finally, all nodes in `MNodes` are marked as visited. This operation is repeated until no unvisited nodes are left in G.

The process of creating superedges is detailed in `ConnectNodes` procedure. It iterates for all supernodes that are created from the previous step to add edges to their neighbors. Using the `NNodes` of $S_i$ and NodeList, the superedge can be created (steps 2-3).

**Procedure** CreateNodes(G)

```
1  Let G_s be empty graph
2  adj = adjlist(G)
3  set i = 0
4  while len(G.UnvisitedNodes) > 0 do
5      i = 0
6      MNodes = FullComponent()
7      NNodes = Neighbors(MNodes)
8      AttList = Attributes(MNodes)
9      S_i = CreateNode(NNode, MNode, AttList)
10     UpdateSupList(NodeList, MaoList)
11     AddNode(G_s, S_i)
12     Mark(MNodes)
13     i = i + 1
14 return G_s
```

**Procedure** ConnectNodes($G_s$)

```
1  forall S_i in G_s do
2      forall k in S_i.NNodes do
3          AddEdge(S_i, G_s.nodes(Nodelist[K]))
4  return G_s
```

**Algorithm 1** Create Summary Graph

**Input :** Original graph G.
**Output:** Summary graph $G_s$.

```
1  Let G_s be an empty graph.
2  G_s = CreateNodes(G)
3  G_s = ConnectNodes(G_s)
4  return G_s
```

### 2) COMMUNITY SEARCH MODULE

This section shows the community search module that leverages the notion of community influence defined in Section IV. Different search algorithms are proposed to identify the top-r k-influential attributed communities. This section first presents the basic exact algorithm `BasicExact` which traces each supernode for all possible solutions (k-cliques, adjacent k-cliques, and k-clique components). This is time-consuming for large graphs. So, a more efficient algorithm is presented called `EnhancedExact` that ignores some of the irrelevant results. Inspired by `EnhancedExact`, an approximate search algorithm is proposed.

#### *a: BASIC EXACT ALGORITHM*

The main idea of the basic exact algorithm is to find all possible communities with the required attributes. Then, it returns the most influential communities according to Eq. 4. The basic exact search algorithm consists of three main steps:

1) *Find All k-Clique Components That Contain the Required Attributes:* First, MapList is used to retrieve all supernodes that contain the required attributes. The supernodes can be considered as seed nodes for creating communities. Each supernode and its neighbors are checked. The Attlist of each supernode is used to identify the subgraph H of the nodes that contain the required attributes. Then, construct an adjacent graph of H and retrieve all communities. In the case no solution is found, nodes that do not have any of the required attributes can be added to H and the test is repeated.
2) *Calculate the Influence of Each Community:* Eq.(4) is used to calculate the influence of each community that resulted from step 1.
   For calculating the diameter: it is time-consuming to calculate the shortest path between all pairs. We can exploit the adjacency graph properties to minimize the calculations between pairs: a) the diameter of a single node returned from the adjacency graph (k-clique), b) the diameter of 2 adjacent k-cliques is equal to 2 because they share (k-1) nodes, c) for k-clique component the shortest path between each node and its indirect nodes is calculated.
3) *Check the Maximal Condition:* Check that for each community $C$ there is no community $C\prime$ such that $C \subset C\prime$, and $f(C\prime_x) >= f(C_x)$

Algorithm 2 works as follows: First, it loops on each supernode $S_i$ in MapList that contains the required attribute. Then, finds the subgraph that contains $S_i$ and its neighbors which contains the effective nodes (step 4). The algorithm ignores irrelevant subgraphs using the condition in step 5. Such that the size of the inducted subgraph $H_2$ should be at least equal to the size of the k-clique. The `Size(`$H_2$`)` uses the following equation to determine the size of $H_2$:

$$Size(H_2) = \sum_{S_j \in N(S_i)} weight(e(S_i, S_j)) + \sum |S_i.MNodes|(|S_i.MNodes| - 1)/2 \quad (5)$$

Then, all possible solutions are returned using `FindAllPossible` procedure that: first, builds the adjacency graph (see definition 2.7) steps (3-9). Based on this adjacency graph the adjacent k-cliques (see definition 2.6) are generated (step 8). Also, k-clique components (see definition 2.8) are generated from the adjacency

graph (steps 10-11). The influence of each community is calculated using equation 4. Then, `CheckMaximal` checks the maximal condition. Finally, the top-r influential communities are returned (step 14).

**Procedure** FindALLPossible(H, k)

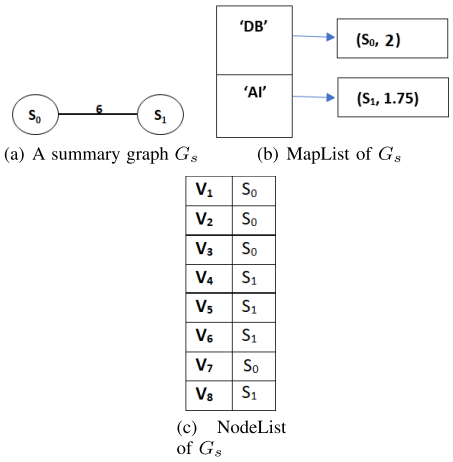
```
1  R1 = { } R2 = { } R3 = { }
2  R1 = FindCliques(H, k) // see Definition 2.5
3  AdjGraph = Graph()
4  AdjGraph.AddNodes(R1)
5  for C1 in R1 do
6      for C2 in FindAdjCliques(C1) do
7          if |C1.intersection(C2) >= (k-1)| then
8              R2.ADD(C1, C2)
9              AdjGraph.AddEdge(C1, C2)
10 for Component in ConnectedComponents(AdjGraph) do
11     R3.Add(Component)
12 Result.ADD(R1, R2, R3)
13 return Result
```

**Algorithm 2** Basic Exact Query Processing

**Input :** r, K, Key, $G_s$
**Output:** Top-r K-Influential Communities.

```
1  Result =
2  forall S_i in MapList(Key) do
3      H_1 = Graph(S_i, Neighbors(S_i))
4      H_2 = Graph(H_1, key)
5      if Size(H_2) ≥ k(k − 1)/2 then
6          R = FindALLPossible(H_2, k) Result.ADD(R)
7      end
8      if Result.ISEmpty() then
9          R = FindALLPossible(H_1, k)
10         Result.ADD(R)
11     end
12     CheckMaximal(Result)
13 end
14 return Top-r(Result)
```

*Example 5.2:* Given an undirected graph G in Figure 1 with attributes [DB, AI]. The query is to find top-2 DB communities with k = 4. Following Eq.2 to calculate the weight of each attribute associated with each node: The weights of attributes associated with $V_1$: The weight of $DB = (1) + (4/4) + (0.5 * 0) = 2$, and the weight of $AI = (0) + (1/4) + (0.5 * 1) = 0.75$. The vector of weights associated with $V_1 = [2, 0.75]$. Both $V_2$ and $V_3$ are the same as $V_1$. The weights of attributes associated with $V_4 = [1, 1.75]$. Also, $V_5$ and $V_6$ are the same as $V_4$. The weights of attributes associated with node $V_7 = [2, 1]$. The weights of attributes associated with node $V_8 = [2, 1.5]$. Then, follow Algorithm 1




(a) A summary graph $G_s$ (b) MapList of $G_s$ (c) NodeList of $G_s$

**FIGURE 3.** The summary graph, MapList, and NodeList of original graph G.

to create the summary graph $G_s$ shown in Figure 3(a). This summary graph contains 2 supernodes $S_0, S_1$, where $S_0 = \{V_1, V_2, V_3, V_7\}$, and $S_1 = \{V_4, V_5, V_6, V_8\}$. The weight of the edge between the two supernodes $E_{0,1} = 6$, where the number of neighbors = 6. Figures 3(b) and 3(c) show the `MapList` and the `NodeList` of $G_s$. The top-2 4-influential communities is $\{V_7, V_1, V_2, V_3, V_8\}$.

### b: ENHANCEDEXACT ALGORITHM

Algorithm 2 requires exploring all possible communities in G that satisfy the query conditions which is a time-consuming operation. The `EnhancedExact` algorithm aims to reduce required steps using some verification rules. Therefore, a large sum of computation can be cut off during the verification. More specifically, there is no need to find all possible cases of adjacent k-cliques nor k-clique components to discover the required influential communities. Only communities with a high influence value should be detected. The *EnhancedExact* algorithm can be summarized as follow:

1) Find all possible k-cliques from each supernode stored in the `MapList` of the required attributes.
2) Store the lists of cliques into a cliques container (linked list) where each node in the linked list contains a list of cliques resulted from a specific supernode. The linked list is sorted according to the influence value of the maximum k-clique in each node of the linked list.
3) In order to obtain the adjacent cliques that represent communities from each list in the container, only the adjacent cliques that contain at least one clique with an influence value greater than the influence values in the next items of the container need to be tested.
4) The same for testing k-clique components: only the series of adjacency k-cliques that contains at least one greater k-clique needs to be tested.

Details are given in Algorithm 3. First, it finds all list of k-cliques by invoking `FindCliques` (step 2) in the

**Procedure** HigherInfCliques(C1, C2)

```
1 HCLiques = []
2 for h in C1 do
3     if inf(h > inf(C2[0]) then
4         HCLiques.Add(h)
5     else
6         break
7 return HCLiques
```

`Container` list where each element in the `Container` is a list of cliques corresponding to a supernode in the `MapList`. Then, for each element i in the `Container`, it finds the k- cliques that have a higher influence value than cliques in the next element in the container by invoking `HigherInfCliques` (step 4) and adding communities in the result set from AdjComm that have influence value> the minimum influence value in maxcliques (step 6) and removing them from AdjComm. It checks the possible adjacent k-cliques and k-clique components in the list i that may have influence > the higher influence value in each list in the rest of the container (steps 9-43). First, it checks if the clique with the minimum influence value in i has an influence value greater than the maximum influence value of cliques in j, it creates an adjacency graph with all cliques in i and returns the adjacent k-cliques and k-clique components from the graph (steps 12-26). Otherwise, it creates an adjacency graph with only cliques in i that have influence value> the clique with the maximum influence value in j and returns the adjacent k-cliques and k-clique components from the graph (steps 29-42).

### c: APPROXIMATE SEARCH ALGORITHM

Recall that in Algorithm 3 the most time-consuming step is to generate all k-cliques. Thus to reduce the time overhead, the approximate algorithm is proposed. Instead of finding all possible k-cliques from the beginning, the approximate algorithm finds the k-cliques from the stored supernodes one by one until the required number of communities is reached. It tests the supernodes according to the weights stored in the MapList. Given the aforementioned Eq.2 in which weight is calculated, the effect of the node, its neighbors, and the next neighbors are considered. Consequently, there is a high possibility that the supernode with the largest weight to return communities with high influence value.

First, it finds the k-cliques that can be reached from the first supernode $S_i$ stored in the MapList. Then, it finds the k-cliques for the second supernode $S_j$ stored in the MapList and stores the k-cliques `MaxCliques` resulted from $S_i$ and have influence value greater than the cliques resulted from $S_j$. Then, it finds the adjacent k-cliques and k-clique components that contain any clique in `MaxCliques`. This operation is repeated until finding the top-r solutions.

**Algorithm 3** EnhancedExact Query Processing Algorithm

**Input :** r, K, Key, $G_s$
**Output:** Top-r K-Influential Communities.

```
1  Result = {}; AdjComm = {}
2  CliqueContainer = FindCliques(MapList[Key])
3  for i = 0 to |CliqueContainer| do
4      maxcliques = HigherInfCliques(CliqueContainer[i],
         CliqueContainer[i + 1])
5      Result.Add(maxcliques)
6      Result.Add(MaxComm(AdjComm,
         min(maxcliques)))
7      if |Result| ≥ r then
8          return Result
9      end
10     for j = i + 1 to |CliqueContainer| do
11         AdjGraph = Graph() R1 = {} R2 = {}
12         if MinInf(CliqueContainer[i]) >
             MaxInf(CliqueContainer[j]) then
13             AdjGraph.AddNodes(CliqueContainer[i])
14             for C1 in CliqueContainer[i] do
15                 for C2 in FindAdjCliques(C1) do
16                     if |C1.intersection(C2) >= (k-1)|
                         then
17                         R1.AddCompInf(C1, C2)
18                         AdjGraph.AddEdge(C1, C2)
19                     end
20                 end
21             end
22             for Component in
                 ConnectedComponents(AdjGraph) do
23                 R2.AddCompInf(Component)
24             end
25             AdjComm.Add(R1)
26             AdjComm.Add(R2)
27             break
28         else
29             HInfCliques = HigherInfCliques
                 (CliqueContainer[i], CliqueContainer[j])
30             for C1 in HInfCliques do
31                 for C2 in CliqueContainer[j] do
32                     if C1 share n − 1 nodes with C2
                         then
33                         AdjGraph.Add(C1, C2)
34                         R1.AddCompInf(C1, C2)
35                     end
36                 end
37             end
38             for Component in
                 ConnectedComponents(AdjGraph) do
39                 R2.AddCompInf(Component)
40             end
41             AdjComm.Add(R1, R2)
42         end
43     end
44 end
45 return Result
```

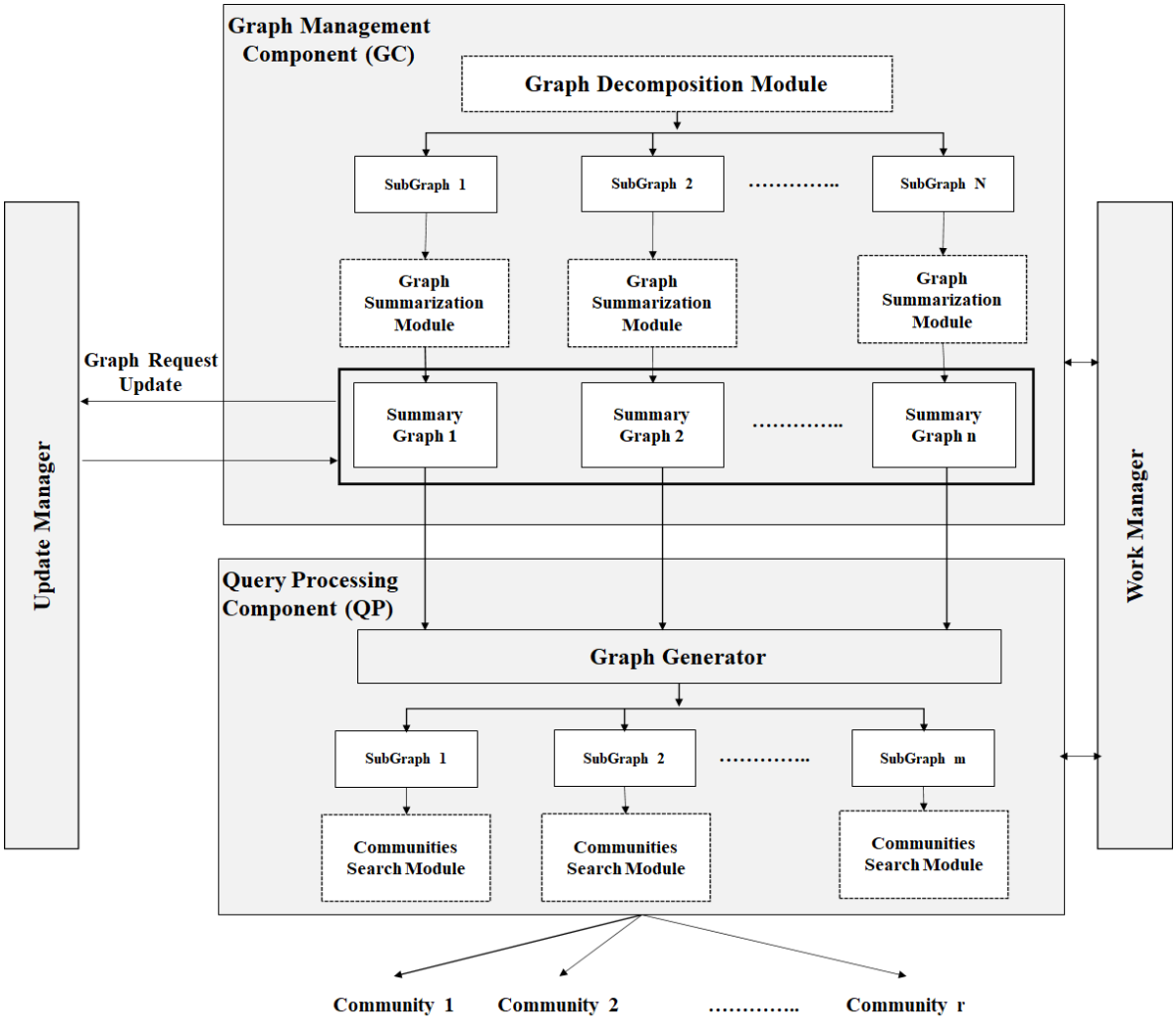



**FIGURE 4.** Decomposed InfACom technique.

### B. DECOMPOSED InfACom

Decomposed InfACom is a parallel technique that employs a graph partitioning technique to improve the performance of the graph summarization module as well as the community search module introduced by the Whole InfACom technique. Figure 4 gives the Decomposed InfACom technique depicting graph management component (GC), query processing component (QP), update manager (UM), and work manager (WM).

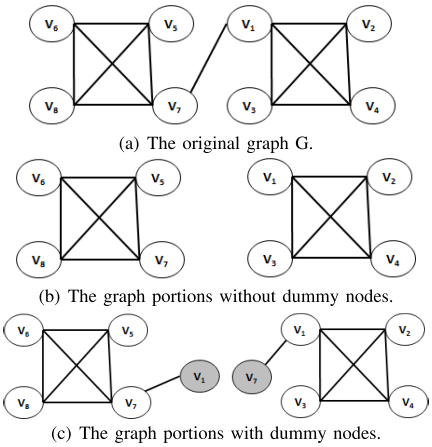
#### 1) GRAPH MANAGEMENT COMPONENT (GC)

This section discusses the main components of the GC: graph decomposition module and graph summarization module.

##### a: GRAPH DECOMPOSITION MODULE

It is a use of the method in reference [4] with no modification. Given a large graph, it is a challenge to employ the summarization module introduced by Whole InfACom. There is a need to decompose the original graph into relevant groups of nodes such that the summarization algorithm (Algorithm 1) can be executed for each partition in parallel. The original graph can be decomposed using the Louvain heuristic method in [4] which is an algorithm for detecting communities in networks. It maximizes a modularity score for each community, where the modularity quantifies the quality of an assignment of nodes to communities by evaluating how much more densely connected the nodes within a community are, compared to how connected they would be in a random network. The main idea in [4] includes two basic steps: 1) modularity optimization: is a greedy assignment of nodes to communities, 2) community aggregation: is the definition of a new coarse-grained network, based on the communities found in the first step. These two steps are repeated until no further modularity-increasing reassignments of communities are possible.

For the initial step of [4], assume that each node of the network belongs to a different community. Then, iterate through each node in the network and remove it from its current community and replace it in the community of one of




(a) The original graph G.

(b) The graph portions without dummy nodes.

(c) The graph portions with dummy nodes.

**FIGURE 5.** **The graph decomposition.**

its neighbors. Then, compute the modularity change for each of the node's neighbors. The node moves into the community where the results in maximal modularity change. If none of these modularity changes are positive, the node stays in its current community. This process is repeated for all nodes until no community assignment changes. In the community aggregation step of the algorithm, a new network is created which its nodes will be the communities resulting from the modularity optimization step.

Applying the algorithm in [4] on the given network decomposed it into different partitions. Nevertheless, there is a need to keep the relationship between these partitions in order to find communities that are distributed across different partitions. So, the proposed algorithm introduces the definition of boundary nodes as nodes that have neighbors in different partitions. Let node i in partition $P_i$ be a boundary node that has neighbors in partition $P_j$. The proposed algorithm creates a dummy node for node i in partition $P_j$.

*Example 5.3:* Given the original graph G in Figure 5(a). After applying Algorithm in [4], we can get two partitions $G_1 = (v_5, v_6, v_7, v_8)$ and $G_2 = (v_1, v_2, v_3, v_4)$ as shown in Figure 5(b). However, Figure 5(c) shows the partitions after applying the dummy nodes $v_1$ and $v_7$.

#### *b: GRAPH SUMMARIZATION MODULE*

This module is responsible for summarizing each subgraph generated from the graph decomposition module in a different thread. This operation can be done by customizing Algorithm 1. In creating supernodes: the graph summarization module ignores the dummy nodes in establishing the MNodes list of the supernodes. The dummy nodes are essential in creating `NNodes` set. Each subgraph has its MapList data structure that maintains attributes along with supernodes that contain each attribute.

### 2) QUERY PROCESSING COMPONENT (QP)

*The Query Processing Component (QP):* Is responsible for retrieving the required communities based on the created summary graphs. `QP` contains both the subgraph generator and the community search module. The subgraph generator has 2 main tasks: 1) generates the subgraphs that contain the required attributes in each summary graph, and 2) generates the distributed subgraphs that contain the required attributes in different summary graphs using the dummy nodes. It works as follows: First, for each supernode $S_i$ in MapList that contains the required attributes, the subgraph generator finds all neighbors of $S_i$. In the case of $S_i$ containing dummy nodes, the subgraph generator tries to find all neighbors of $S_i$ in different summary graphs. Then, the subgraph of $S_i$ and its neighbors is used as input for the community search module. The community search module is responsible for finding the influential communities in the subgraphs returned by the subgraph generator. The BasicExact, EnhancedExact, or Approximate algorithm is executed on these subgraphs in different threads.

### 3) UPDATE MANAGER AND WORK MANAGER (WM)

#### *a: UPDATE MANAGER*

In practice, the graphs are continuously evolving. Thus attributes and edges of graphs are often frequently updated. When the graph is updated, the summary graph also needs to be updated. A straightforward method for handling the dynamic graph is to rebuild the summary graph from scratch when an update is made. However, this method is very inefficient, especially when the updates are very frequent. To alleviate this issue, the update manager is responsible for dynamically maintaining the summary graph efficiently without rebuilding it from scratch. The description is mentioned in detail in Section VI.

#### *b: THE WORK MANAGER (WM)*

WM communicates with both the `GC` and the `QP`. First, `WM` maintains a thread pool such that the summarization module can be executed for different subgraphs result from the decomposition module in different threads. Also, `WM` opens different threads for the community creation module to execute different summary graphs. Once the execution of the summarization module or the communities creation module in any thread is finished, WM returns that thread to the thread pool so that it can be used to handle another request. Maintaining a thread pool permits fast request handling without the overhead of thread creation. WM oversees the execution of all requests until their end.

## VI. INFLUENTIAL ATTRIBUTED COMMUNITY (InfACom) IN DYNAMIC NETWORKS

In this section, we consider the problem of managing dynamic networks where edges and/or attributes could be inserted or deleted. Section VI-A presents how to handle edge insertion. Section VI-B discusses how to handle edge deletion. Finally, section VI-C shows how to handle attribute updates.

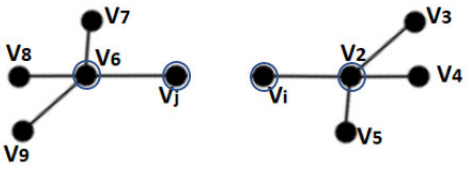



**FIGURE 6.** Original graph G.

### A. HANDLING EDGE INSERTION

Each node in the original graph exists in only one supernode in the summary graph. As explained in Section V-A1, a supernode has three components MNodes, NNodes; for a list of its neighbors, and the third component AttList; the third component the AttList, where each attribute has a list of nodes containing the attribute. The insertion of an edge between node $v_i$ and $v_j$ triggers updates in these components.

The attributes weight associated with $v_i$, $v_j$, and their neighbors will be affected according to the use of Eq. 2 that depends on the direct neighbors and next neighbors of each node. Also, the NNodes of $v_i$ and $v_j$ will be updated by adding $v_j$ to the neighbors of $v_j$ and vice versa.

To reflect the edge insertion on the summary graph, the supernodes that contain $v_i$ and $v_j$ will be changed as well as the supernodes that contain the neighbors of $v_i$ and $v_j$. The steps of an edge insertion can be summarized as follow:

1) Determine the supernodes $S_i$ and $S_j$ that contain $v_i$ and $v_j$ using the NodeList.
2) Update the `NNodes` of both $S_i$ and $S_j$ by adding $v_j$ to the neighbors of $v_i$ and vice versa.
3) Update the attributes weight associated with $v_i$ and $v_j$.
4) Update attributes weight of the neighbors of $v_i$ and $v_j$. First, use the `NNodes` sets of $v_i$ and $v_j$ to determine their neighbors. Then, use NodeList to determine the supernodes of these neighbors. Finally, update the attributes weight using Eq. 2.
5) Add superedge between $S_i$ and $S_j$ and update the edge weight.

Example 6.1 shows how the edge insertion between 2 nodes will affect the attribute weights associated with these 2 nodes as well as the neighbors of them.

*Example 6.1:* Figure 6 shows the original graph G. Adding the edge ($v_i$, $v_j$) will affect the attributes weight of the nodes $v_i$ and $v_j$ as well as $v_2$ and $v_6$. Regarding $v_i$, all attributes associated with it will be affected by the nodes $v_j$ (the neighbor of $v_j$), and $v_6$ (the next neighbor). Regarding $v_j$, all attributes associated with it will be affected by the nodes $v_i$ (the neighbor of $v_i$), and $v_2$ (the next neighbor). Regarding $v_2$, all attributes associated with it will be affected by the nodes $v_j$ (the next neighbor). Regarding $v_6$, all attributes associated with it will be affected by the nodes $v_6$ (the next neighbor).

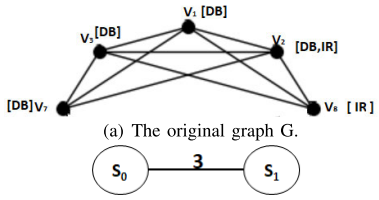
*Merging Two Supernodes:* The insert of an edge may trigger the request of merging two different supernodes $S_i$ and $S_j$ into one. This can be done when the merge of $S_i$ and $S_j$ forms a clique (all nodes in MNodes of $S_i$ are connected to all nodes MNodes of $S_j$).

The size of the clique is measured by $n(n-1)/2$ where n is the number of nodes. To ensure that the merge of $S_i$ and $S_j$ became a clique, its size should be $n(n-1)/2$. The merge is done if Eq.(6) is satisfied:

$$[Size(S_i) + Size(S_j)] + Weight(Edge(S_i, S_j)) = \frac{n(n-1)}{2} \tag{6}$$

where, $n = |S_i.MNodes| + |S_j.MNodes|$, and $Size(S_i) = |S_i.MNodes| * (|S_i.MNodes| - 1)/2$ (because $S_i$ is a clique).

*Example 6.2:* Figure 7 shows the merge case. The original graph G is shown in Figure 7(a), and the summary graph of G is shown in Figure 7(b) where the nodes $S_0 = \{v_1, v_2, v_3, v_7\}$, $S_1 = \{v_8\}$. Assume that an edge e($v_7$, $v_8$) will be added to G. After inserting the edge, we find that every node in $S_0$ is connected to every node in $S_1$. In this case, $S_0$ and $S_1$ should be merged into one supernode.




(a) The original graph G.

(b) The summary graph of G.

**FIGURE 7.** Merging two supernodes.

Algorithm 4 illustrates the details of inserting an edge e($v_i$, $v_j$). First, it determines the supernodes that contain $v_i$, and $v_j$ and obtains $S_i$, $S_j$ (steps 1-2). If Eq.(6) is satisfied then `Merge` is triggered to merge $S_i$, and $S_j$ into one supernode and removes $S_j$ from the summary graph as well as updates the NodeList data structure. Otherwise, it updates the neighbor sets of $S_i$, $S_j$ to append $v_j$, $v_i$ (steps 6-7). Next, `Calweight` updates the edge weight between $S_i$, and $S_j$ by adding 1 that is resulted from the new edge. Finally, `UpdateNeighbors` is triggered to update superedges (step 9).

**Procedure** Merge($G_s$, $S_i$, $S_j$)

1 ExtendNodes($S_i$, $S_j$)
2 ExtendNeighbors($S_i$, $S_j$)
3 AttList($S_i$, $S_j$)
4 UpdateNeighbors($S_i$)
5 Remove($S_j$)
6 UpdateNodeList()

### B. HANDLING EDGE DELETION

Similar to the edge insertion, removing an edge between $v_i$ and $v_j$ triggers the updates to the summary graph. This section discusses different cases for removing an edge.

**Algorithm 4** Maintain Edge Insertion

**Input** : $G_s$, $v_i$, $v_j$, NodeList
**Output:** $G_s$

1 Let $S_i$ = NodeList[$v_i$]
2 Let $S_j$ = NodeList[$v_j$]
3 $G_s$[$Si$.NNodes][$v_i$].add($v_j$)
4 $G_s$[$Sj$.NNodes][$v_j$].add($v_i$)
5 UpdateAttWeight()
6 UpdateSuperEdgeWeight($S_i$, $S_j$)
7 **if** *Eq.(6)* **then**
8 | Merge($G_s$, $S_i$, $S_j$)
9 **end**
10 **return** $G_s$

*Case 1:* Removing an edge between two nodes that coexist in the same supernode. This case requires splitting the supernode into two different supernodes because the nodes in MNodes list now do not form a clique. Each new supernode should contain a fully connected component.

Consider the original graph G in Figure 7(a), and its summary in Figure 7(b). If an edge e($V_1$, $V_2$) is deleted. First, a new node $S_N$ is created with node $V_1$. Then, the metadata of $S_N$ (NNodes, Attlist) is added. Add the neighbors of $V_1$ which are {$V_3$, $V_{11}$} to $S_N$.NNodes. Also, the AttList with 'DB' attribute is created and calculate the weight using Eq. 2. Next, $V_1$ is removed from the MNodes of the supernode $S_0$.

*Case 2:* Removing an edge between two nodes existing in different supernodes. Suppose that an edge e($v_i$, $v_j$) is removed from the original graph and $S_i$ contains $v_i$, $S_j$ contains $v_j$. In this case, $v_i$ will be removed from the NNodes of $S_j$ as well as $v_j$ will be removed from the NNodes of $S_i$. Finally, the superedge between $S_i$ and $S_i$ will be maintained (deleted if there is no link between them otherwise subtract 1 from the weight).

For both cases 1 and 2, the weight of the attributes associated with $v_i$ and $v_j$ as well as their neighbors should be maintained.

Algorithm 5 illustrates the details of removing an edge e($v_i$, $v_j$). Case 1 is illustrated in (steps 4-10). First, a new supernode $S_N$ is created (step 4). Then, node $v_i$ is inserted in $S_N$ and the metadata of $S_N$ is created (steps 7-8). Such that the NNodes of $S_N$ is set to the neighbors of $v_i$ in $S_j$ as well as the NNodes of $S_j$ except $v_i$. Next, $S_N$ is added to the summary graph (step 9) and $v_i$ is removed from $S_i$. Finally, `UpdateNeighbors` maintains the superedges of the neighbors of $S_N$, and $S_i$(step 10). Case 2: is illustrated in (steps 12-14). Such that $v_i$, $v_j$ are removed from the MNodes of $S_j$, and $S_i$ (steps 12-13). Finally, `UpdateNeighbors` updates the supernodes between $S_j$, and $S_i$ and their neighbors by subtracting 1 from edge weight.

**Algorithm 5** Maintain Edge Deletion

**Input** : $G_s$, $v_i$, $v_j$, NodeList
**Output:** $G_s$

1 Let $S_i$ = NodeList[$v_i$]
2 Let $S_j$ = NodeList[$v_j$]
3 **if** $S_i = S_j$ **then**
4 | Let $S_N$ is a new supernode.
5 | $S_N$.MNodes = $v_i$
6 | Remove($S_i$, $v_i$)
7 | $S_N$.NNodes = $S_i$.NNodes[$v_i$]
8 | $S_N$.AttList = GetAttList($S_i$, $v_i$)
9 | UpdateATTWeight()
10 | $G_s$.AddNode($S_N$)
11 | UpdateNeighbors($G_s$, $S_i$, $S_N$)
12 **else**
13 | $S_i$.NNodes[$v_i$].Remove($v_j$)
14 | $S_j$.NNodes[$v_j$].Remove($v_i$)
15 | UpdateATTWeight()
16 | UpdateNeighbors($G_s$, $S_i$, $S_j$)
17 **end**
18 **return** $G_s$

### C. HANDLING ATTRIBUTE UPDATE

This section discusses how to handle removing and inserting an attribute $a_i$ associated with a specific node $v_i$. In the case of inserting an attribute, the change mainly occurs to the attribute weights of $v_i$ as well as the neighbors and next neighbors of $v_i$ that contain $a_i$. Also, the Maplist should be updated by adding the supernode $S_i$ that contains $v_i$ to the entry of $a_i$.

The weight of $a_i$ associated with $v_i$ is maintained by adding 1 (the value of $\alpha$ in Eq. 2). Then, the change in weight of $a_i$ associated with the neighbors of $v_i$ that contain $a_i$ is affected by only one extra node. It can be updated by adding $\Delta w_1$ where

$$\Delta w_1 = 1/(|deg(n)|), \quad \forall n \in neighbors(V_i)$$

Finally, for the next neighbors of $v_i$ that contain $a_i$, those nodes are also affected by only one extra node which is not a direct neighbor. It can be updated by adding $\Delta w_2$ where

$$\Delta w_2 = 0.5/(|n'|)$$

where n' is the set of next neighbors of $n$.

**Procedure** AddAtrr($G_s$, $v_i$, att, NodeList, MapList)

1 Let $S_i$ = NodeList[$v_i$]
2 $S_i$.AttList.add(att, $v_i$)
3 UpdateAttWeight()
4 MapList[att].add($S_i$)

Removing an attribute is handled as insertion but the attribute weight associated by the neighbors of $v_i$ is updated

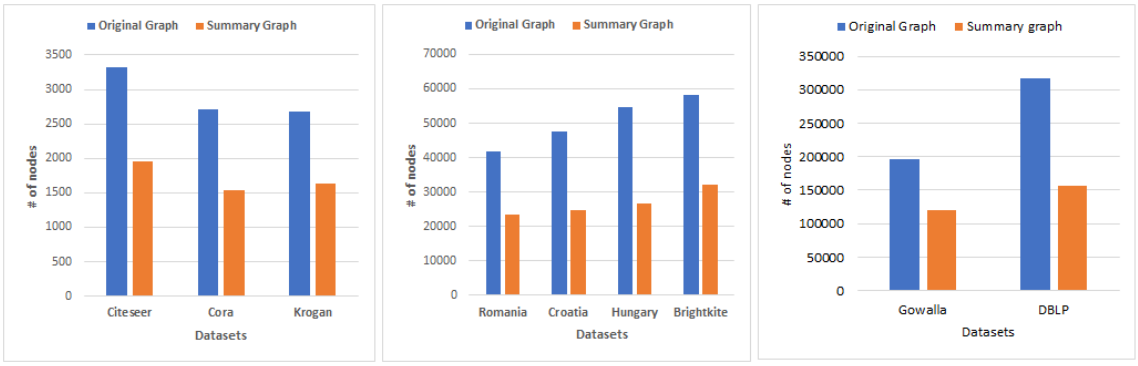



**FIGURE 8.** The summary graphs size for different datasets.

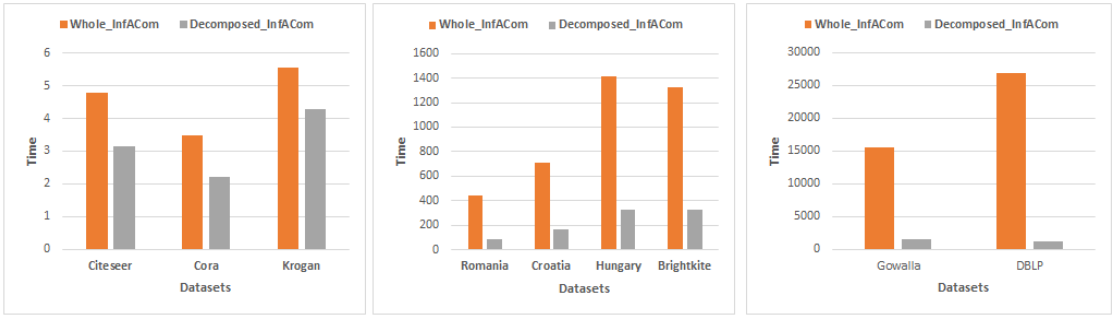



**FIGURE 9.** The summary graphs construction time for different datasets.

by subtracting $\Delta w_1$ and update the attribute weight of the next neighbors of $v_i$ by subtracting $\Delta w_2$.

`AddAtrr` Procedure gives the details of insert a new attribute for a node $v_i$. First, it obtains the supernode of $v_i$ and stores it in $S_i$. Then, update the `AttList` of $S_i$ by adding $v_i$ to att step (3). `UpdateAttWeight` Procedure updates the attribute weight using $\Delta w_1$ and $\Delta w_2$. Finally, Maplist should be maintained by removing $S_i$ from the att entry (step 5). Also, `RemoveAtrr` Procedure shows the details of removing an attribute from the node $v_i$.

**Procedure** RemoveAtrr($G_s$, $v_i$, att, NodeList, MapList)

1 Let $S_i$ = NodeList[$v_i$]
2 $S_i$.AttList.remove(att, $v_i$)
3 UpdateAttWeight()
4 MapList[att].remove($S_i$)

## VII. EXPERIMENTAL RESULTS

This section presents the experimental results that measure the efficiency of our proposed implementations. The proposed algorithms are implemented in Python and all experiments are conducted on Windows 10 with Intel(R) Core(TM) i7 CPU and a 16GB RAM. In all experiments, the summary graph is resident in memory.

**TABLE 1.** Datasets description.

| Dataset | Number of nodes | Number of edges |
|---|---|---|
| Krogan | 2674 | 7075 |
| Cora | 2708 | 5278 |
| Citeseer | 3327 | 4614 |
| Brightkite | 58228 | 214078 |
| Hungary | 54573 | 498202 |
| Romania | 41773 | 125826 |
| Croatia | 47538 | 222887 |
| DBLP | 317080 | 1049866 |
| Gowalla | 196591 | 950327 |

### A. DATASETS

Experimental studies are conducted on real datasets in Table 1. The first dataset is the protein-protein interaction (PPI) network Krogan related to the yeast saccharomyces cerevisiae [22]. Cora and Citeseer datasets are downloaded from (https://linqs.soe.ucsc.edu/data). All other datasets are downloaded from (https://snap.stanford.edu/data).

### B. MEMORY SPACE AND TIME COST EVALUATION FOR SUMMARY GRAPH CONSTRUCTIONS

We build the summary graph for different datasets are shown in Table 1 using both Whole InfACom and Decomposed

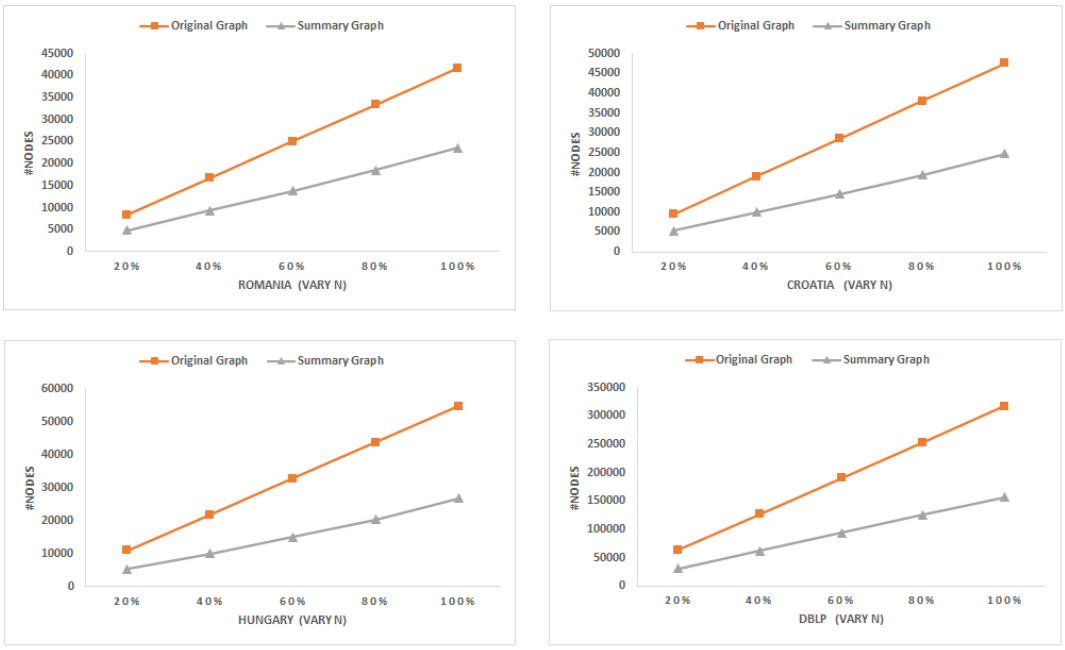



**FIGURE 10.** Scalability testing for the size of the summary graph.

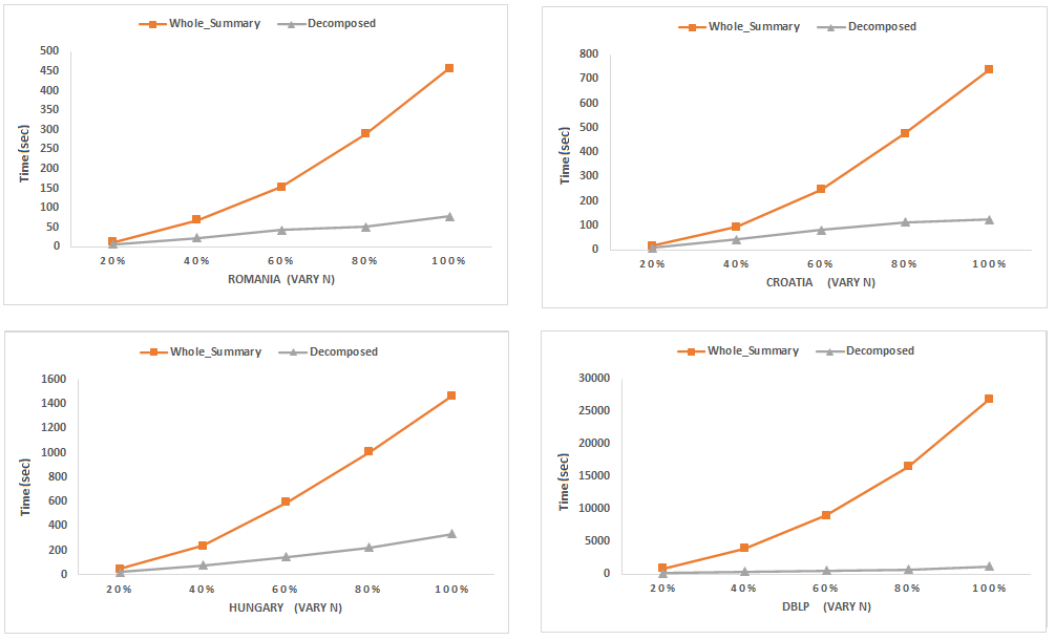



**FIGURE 11.** Scalability testing for the time of summary graph construction.

InfACom techniques. The summary graph construction time is shown in Figure 9. As shown in the result, the time required to partition the original graph and summarize different partitions in parallel with each other requires a lower time than dealing with it as a whole in all datasets. We further compare the number of nodes of the summary graph with the number of nodes of the original graph. The results are depicted in Figure 8. Overall the datasets, the number of nodes of the summary graph is smaller than the number of nodes of the original graph.

### C. SCALABILITY FOR THE SUMMARY GRAPH

In this experiment, we vary the number of nodes in different datasets to study the scalability of the summary graph construction algorithms in Whole InfACom and Decomposed InfACom. The results are shown in Figure 10. As can be seen, both InfACom and Decomposed InfACom scale near linearly in most datasets. Moreover, we can see that Decomposed InfACom is faster than the Whole InfACom, which is consistent with the previous observations. In addition, also the scalability results for summary graph size are shown

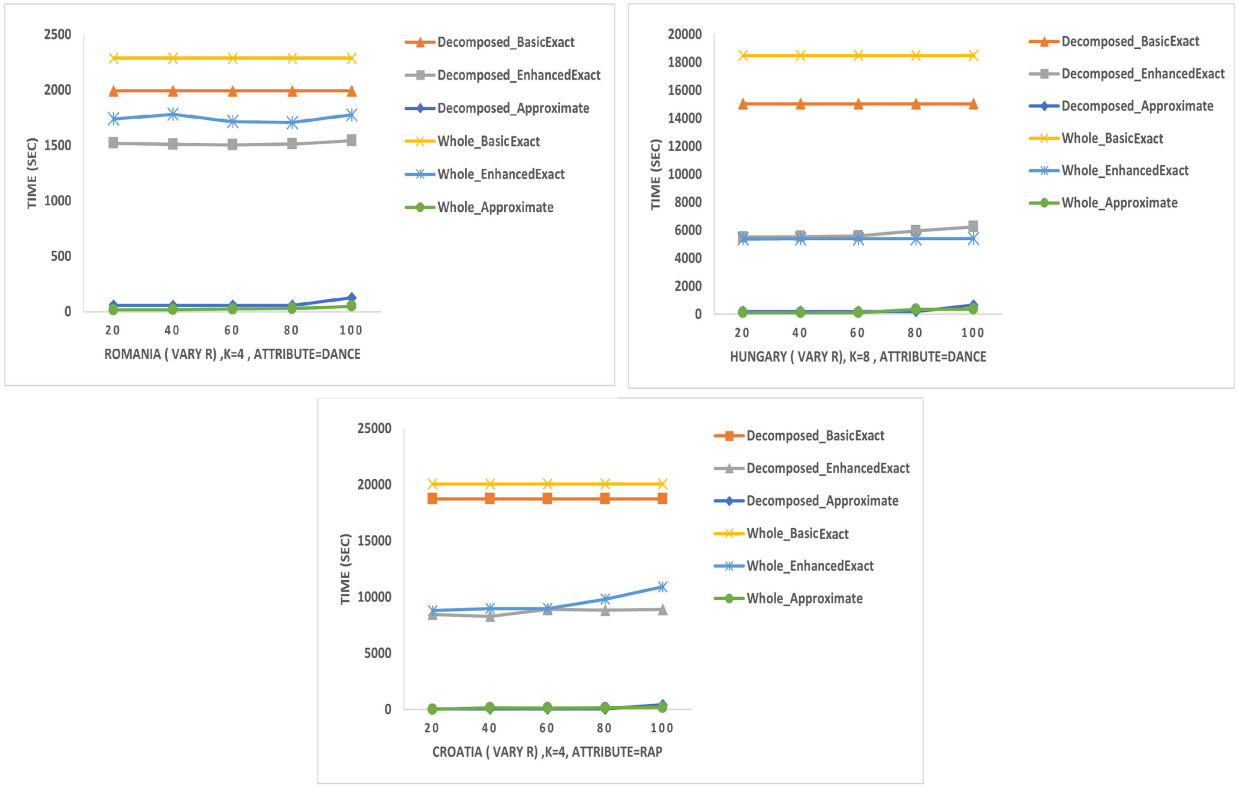



**FIGURE 12.** Query processing time vary r.

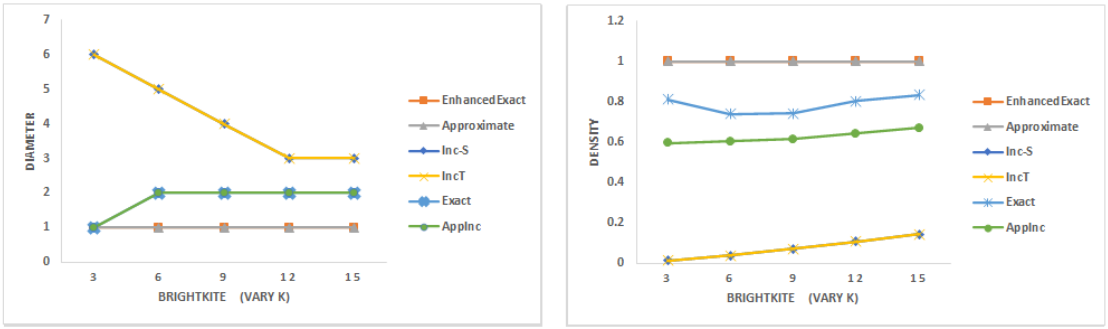



**FIGURE 13.** The structure cohesiveness of the retrieved communities with varying k.

in Figure 11. We can see that the summary graph size is smaller than the original graph size for overall testing cases.

### D. QUERY PROCESSING (VARY r)

We vary the parameter r from 20 to 100 and evaluate the query processing time of the proposed algorithms. The results are shown in Figure 12. For BasicExact and EnhancedExact, the processing time is high. This is because, for both BasicExact and EnhancedExact, the dominant cost is spent on building the k-cliques. The most expensive process is `FindCliques`. In Decomposed InfACom, FindCliques is executed in parallel on different summary graphs. Thus, BasicExact and EnhancedExact in Decomposed InfACom outperform BasicExact and EnhancedExact in Whole InfACom.

For Approximate, when r is small, the processing time increases slowly. However, when r is large, the processing time of Approximate increases. The reason is that when r increases, the size of the finding more k-cliques tends to increase. We can also note that in Whole InfACom, Approximate outperforms Approximate in Decomposed InfACom. The reason is that in Whole InfAcom, when the algorithm stopped when reaches the top-r communities immediately. However, in Decomposed InfACom, the result set needs to be maintained from different threads. Hence, the result set may be filled with communities from the summary graph that does

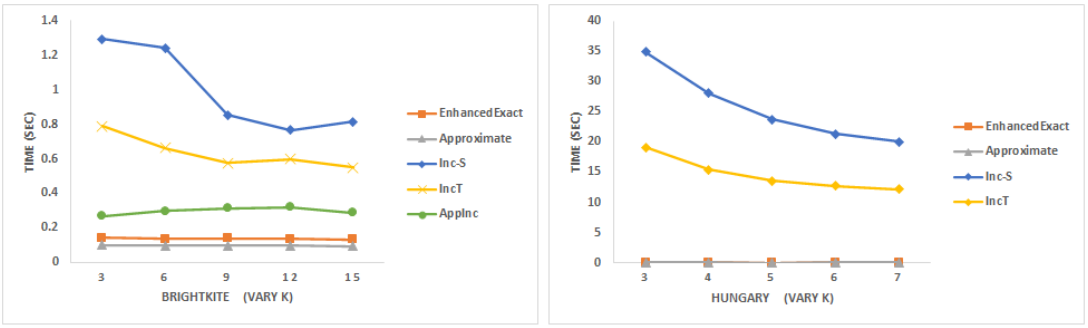



**FIGURE 14.** The structure cohesiveness of the retrieved communities with varying k.

**TABLE 2.** Dynamic updates.

| Dataset | InEdge | DelEdge | IntAttr | DelAttr |
|---|---|---|---|---|
| Hungary | 22.01 | 23.9 | 0.099 | 0.102. |
| Romania | 2.8 | 2.8 | 0.0009 | 0.0009. |
| Croatia | 3.6 | 3.6 | 0.002 | 0.002. |

**TABLE 3.** Processing time in minutes for exact algorithm using Brightkite vary (k).

| K | 3 | 6 | 9 | 12 | 15 |
|---|---|---|---|---|---|
| Exact | 26.4 | 280.4 | 305.1 | 485.8 | 558.6 |

not have more influential communities than another summary graph. So, the algorithm stops executing for a specific summary graph once it returns a community with a smaller influence value than all communities in the result set which has r communities.

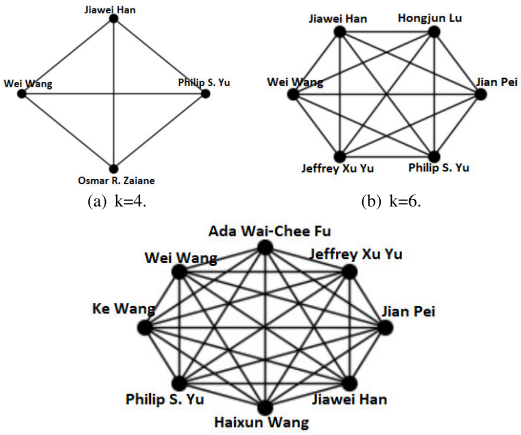
### E. DYNAMIC UPDATE

This experiment shows the evaluation of the updating algorithms. For each dataset, we randomly insert 1000 attributes and after each insertion, the summary graph is updated. Then, we delete the same 1000 attributes and update the summary graph after each deletion. Also, the edge insertion/deletion is evaluated for each dataset. 1000 edges are randomly inserted and after each insertion, the summary graph is updated. Then, we delete the same 1000 edges and update the summary graph after each deletion. The average update time per attribute/edge insertion/deletion is shown in table 2.

### F. COMPARISON WITH STATE-OF-THE-ART

To ensure the accuracy and efficiency of the proposed algorithms, we compared the performance of the proposed algorithms with the approaches presented in [17], [18] as the state-of-the-art approaches as they find the community related to a query vertex in an attributed graph.

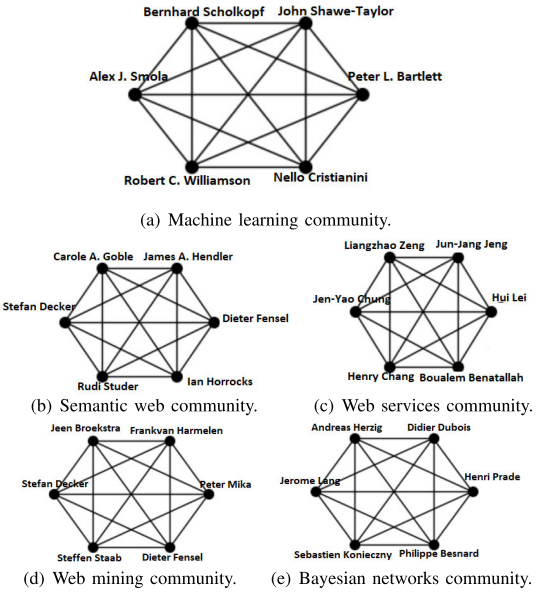
We customized the proposed algorithms, *EnhancedExact*, and *Approximate* algorithms to make them comparable to both *Exact*, and *AppInc* in [17], $Inc-S$, and $Inc-T$ in [18] by searching for the most influential community associated with a specific query vertex (node). We chose 120 query vertices at random with core numbers of 20 or more. A meaningful community containing the query vertex is ensured by such




(a) k=4. (b) k=6.

(c) k=8.

**FIGURE 15.** The most influential DM communities for "Wei Wang" with different k.

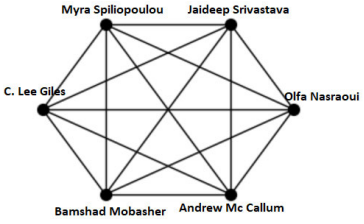



(a) Machine learning community.

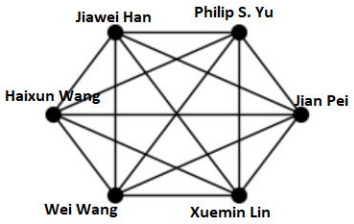
(b) Semantic web community. (c) Web services community.

(d) Web mining community. (e) Bayesian networks community.

**FIGURE 16.** The most influential communities for different research areas with k = 6.

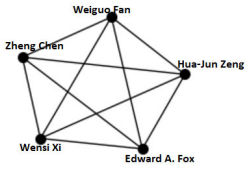



(a) {Machine Learning, Web Mining}

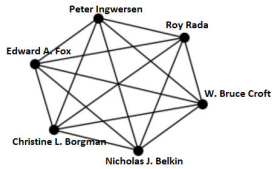



(b) {Database, Data Mining}

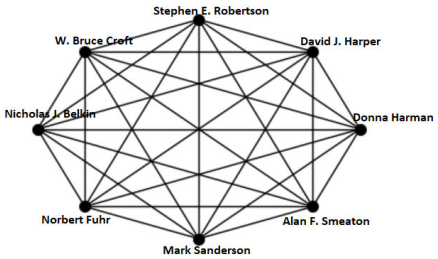
**FIGURE 17.** The most influential for multiple attributes communities with k = 6.



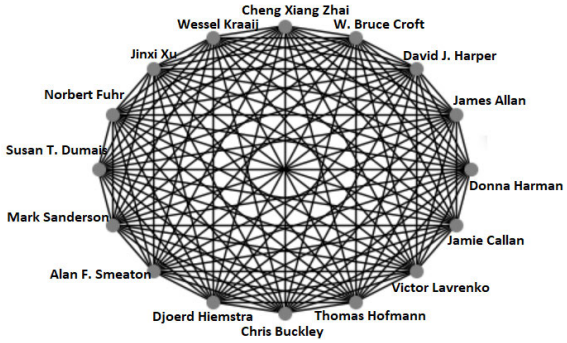

(a) k=5.




(b) k=6.




(c) k=8.

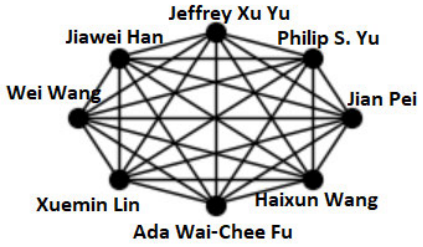



(d) k=14.

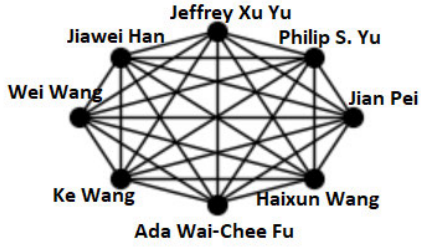
**FIGURE 18.** The most IR influential communities with different k.

a core number constraint. Each data point represents the average result for these 120 queries.

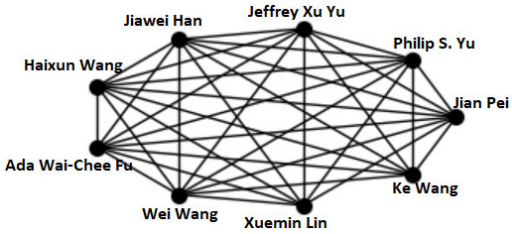
The structure cohesiveness of the retrieved communities is shown in Figure 13. We use popular structural cohesiveness metrics diameter, and density to measure the quality of these communities. Figure 13 shows that the density and the diameter of the retrieved communities of both *EnhancedExact*, and *Approximate* are equal to 1 for all values of k. As the most




(a) Community C1.




(b) Community C2.




(c) Community C3.

**FIGURE 19.** Different three database communities for both 'Wei Wang' and 'Philip S. Yu' with k = 8.

influential communities of these algorithms are k-cliques. For other algorithms, the density increases with the increase in the value of k and the diameter decreases with the increase in the value of k.

Figure 14 shows the query processing times of all algorithms. *EnhancedExact*, and *Approximate* algorithms outperform *AppInc*, *IncT*, and *IncS*. Table 3 shows the processing time in minutes for executingthe *Exact* algorithm. The result shows that the proposed approaches are faster compared to the *Exact* algorithm.

## VIII. CASE STUDIES

We use a topic-coauthor dataset extracted from (http://arnetminer.org) which consists of 8 topics (Data Mining, Web Services, Bayesian Networks, Web Mining, Semantic Web, Machine Learning, Database Systems, and Information Retrieval). The dataset has 5114 nodes and edges. Due to the properties of this dataset, the influence values of the communities are almost identical. So, the results are ranked using the average weight associated with the dataset.

### A. RESULTS FOR RESEARCH AREAS FOR A SPECIFIC AUTHOR WITH DIFFERENT k

Finding meaningful communities that a query vertex belongs to. As we can answer the query ''Find the most influential communities of a specific author''. Figure 15 shows the most

influential Data Mining community that containing the author ''Wei Wang'' with different k.

### B. RESULTS FOR DIFFERENT RESEARCH AREAS

Finding meaningful communities for a research area. As we can answer the query ''Find the most influential communities of a specific research area''. Figure 16 shows the most influential community for different research areas with $k = 6$.

### C. RESULTS FOR MULTIPLE RESEARCH AREAS

Figure 17 shows different communities for multiple research areas with $k = 6$.

### D. RESULTS FOR RESEARCH AREAS WITH DIFFERENT k

Finding meaningful communities for a research area. As we can answer the query ''Find the most influential communities of a specific research area''. Figure 18 shows the most influential Information Retrieval community with different k.

### E. RESULTS FOR MULTIPLE AUTHORS

Find different communities that contain different authors. Figure 19 shows different three database communities that contain both {'Wei Wang', 'Philip S. Yu'} with $k = 8$.

## IX. CONCLUSION AND FUTURE WORK

The problem of community search over very large graphs is fundamental problem in graph analysis. However certain applications require finding the top-r influential communities in the network. This paper discusses different factors that affect the influence of the community. Based on these factors, different Influential Attributed Community (InfACom) implementations based on the concept of k-clique are introduced. Two techniques are presented one for sequential implementation with three variations and one for parallel implementation. We further proposed efficient algorithms for maintaining the InfACom on dynamic graphs. Finally, we present experimental results that show the efficiency of the proposed implementations. For future work, we will study how to use graph pattern matching techniques in identifying influential communities. Also, isomorphism algorithms will be extended to solve the problem. Finally, multi-machine parallelization will be used for enhancing the processing of the proposed implementations.

## REFERENCES

[1] E. Akbas and P. Zhao, ''Truss-based community search: A truss-equivalence based indexing approach,'' *Proc. VLDB Endowment*, vol. 10, no. 11, pp. 1298–1309, Aug. 2017.

[2] N. Barbieri, F. Bonchi, E. Galimberti, and F. Gullo, ''Efficient and effective community search,'' *Data Mining Knowl. Discovery*, vol. 29, no. 5, pp. 1406–1433, Sep. 2015.

[3] F. Bi, L. Chang, X. Lin, and W. Zhang, ''An optimal and progressive approach to online search of top-k influential communities,'' *Proc. VLDB Endowment*, vol. 11, no. 9, pp. 1056–1068, May 2018.

[4] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, ''Fast unfolding of communities in large networks,'' *J. Stat. Mech., Theory Exp.*, vol. 2008, no. 10, Oct. 2008, Art. no. P10008.

[5] H. Cai, V. W. Zheng, and K. C. Chang, ''A comprehensive survey of graph embedding: Problems, techniques, and applications,'' *CoRR*, vol. abs/1709.07604, pp. 1–20, Feb. 2017.

[6] J. Chandran and V. M. Viswanatham, ''Evaluating the effectiveness of community detection algorithms for influence maximization in social networks,'' in *Proc. Int. Conf. Adv. Electr., Comput., Commun. Sustain. Technol. (ICAECT)*, Feb. 2021, pp. 1–11.

[7] L. Chang, W. Li, X. Lin, L. Qin, and W. Zhang, ''PSCAN: Fast and exact structural graph clustering,'' in *Proc. IEEE 32nd Int. Conf. Data Eng. (ICDE)*, May 2016, pp. 253–264.

[8] S. Chen, R. Wei, D. Popova, and A. Thomo, ''Efficient computation of importance based communities in web-scale networks using a single machine,'' in *Proc. 25th ACM Int. Conf. Inf. Knowl. Manage.*, New York, NY, USA, Oct. 2016, pp. 1553–1562.

[9] Y. Chen, Y. Fang, R. Cheng, Y. Li, X. Chen, and J. Zhang, ''Exploring communities in large profiled graphs,'' *IEEE Trans. Knowl. Data Eng.*, vol. 31, no. 8, pp. 1624–1629, Aug. 2019.

[10] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu, ''Efficient core decomposition in massive networks,'' in *Proc. IEEE 27th Int. Conf. Data Eng.*, Apr. 2011, pp. 51–62.

[11] S. Chobe and J. Zhan, ''Advancing community detection using keyword attribute search,'' *J. Big Data*, vol. 6, no. 1, pp. 1–33, Dec. 2019.

[12] J. Clement. (Aug. 2019). *Twitter: Number of Monthly Active Users 2010-2019*. [Online]. Available: https://www.statista.com/statistics/282087/number-of-monthly-active-twitter-users/

[13] W. Cui, Y. Xiao, H. Wang, Y. Lu, and W. Wang, ''Online search of overlapping communities,'' in *Proc. Int. Conf. Manage. Data (SIGMOD)*, New York, NY, USA, 2013, pp. 277–288.

[14] W. Cui, Y. Xiao, H. Wang, and W. Wang, ''Local search of communities in large graphs,'' in *Proc. Int. Conf. Manage. Data (SIGMOD)*, New York, NY, USA, Jun. 2014, pp. 991–1002.

[15] J. Edachery, A. Sen, and F. J. Brandenburg, ''Graph clustering using distance-k cliques,'' in *Graph Drawing*, J. Kratochvíyl, Ed. Berlin, Germany: Springer, 1999, pp. 98–106.

[16] Facebook. (Oct. 2020). *Facebook Reports Third Quarter 2020 Results*. [Online]. Available: https://www.prnewswire.com/news-releases/facebook-reports-third-quarter-2020-results-301163373.html

[17] Y. Fang, R. Cheng, X. Li, S. Luo, and J. Hu, ''Effective community search over large spatial graphs,'' *Proc. VLDB Endowment*, vol. 10, no. 6, pp. 709–720, Feb. 2017.

[18] Y. Fang, R. Cheng, S. Luo, and J. Hu, ''Effective community search for large attributed graphs,'' *Proc. VLDB Endowment*, vol. 9, no. 12, pp. 1233–1244, Aug. 2016.

[19] B. Farzad, O. Pichugina, and L. Koliechkina, ''Multi–layer community detection,'' in *Proc. Int. Conf. Control, Artif. Intell., Robot. Optim. (ICCAIRO)*, May 2018, pp. 133–140.

[20] S. Fortunato, ''Community detection in graphs,'' *Phys. Rep.*, vol. 486, nos. 3–5, pp. 75–174, 2010.

[21] T. Heimo, J. Saramäki, J.-P. Onnela, and K. Kaski, ''Spectral and network methods in the analysis of correlation matrices of stock returns,'' *Phys. A, Stat. Mech. Appl.*, vol. 383, no. 1, pp. 147–151, Sep. 2007.

[22] A. L. Hu and K. C. C. Chan, ''Utilizing both topological and attribute information for protein complex identification in PPI networks,'' *IEEE/ACM Trans. Comput. Biol. Bioinf.*, vol. 10, no. 3, pp. 780–792, May 2013.

[23] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu, ''Querying k-truss community in large and dynamic graphs,'' in *Proc. Int. Conf. Manage. Data (ACM SIGMOD)*, New York, NY, USA, Jun. 2014, pp. 1311–1322.

[24] X. Huang, L. V. S. Lakshmanan, and J. Xu, ''Community search over big graphs: Models, algorithms, and opportunities,'' in *Proc. IEEE 33rd Int. Conf. Data Eng. (ICDE)*, Apr. 2017, pp. 1451–1454.

[25] X. Huang and L. V. S. Lakshmanan, ''Attribute-driven community search,'' *Proc. VLDB Endowment*, vol. 10, no. 9, pp. 949–960, May 2017.

[26] X. Huang, L. V. S. Lakshmanan, J. X. Yu, and H. Cheng, ''Approximate closest community search in networks,'' *Proc. VLDB Endowment*, vol. 9, no. 4, pp. 276–287, Dec. 2015.

[27] X. Huang, W. Lu, and L. V. S. Lakshmanan, ''Truss decomposition of probabilistic graphs: Semantics and algorithms,'' in *Proc. Int. Conf. Manage. Data*, New York, NY, USA, Jun. 2016, pp. 77–90.

[28] V. Jagadishwari and V. Umadevi, ''Empirical analysis of community detection algorithms,'' in *Proc. 2nd Int. Conf. Electr., Comput. Commun. Technol. (ICECCT)*, Feb. 2017, pp. 1–5.

[29] P. F. Jonsson and P. A. Bates, ''Global topological features of cancer proteins in the human interactome,'' *Bioinformatics*, vol. 22, no. 18, pp. 2291–2297, Jul. 2006.

[30] B. S. Khan and M. A. Niazi, "Network community detection: A review and visual survey," *CoRR*, vol. abs/1708.00977, pp. 1–39, Aug. 2017.
[31] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo, "K-core decomposition of large networks on a single PC," *Proc. VLDB Endowment*, vol. 9, no. 1, pp. 13–23, Sep. 2015.
[32] J. Li, X. Wang, K. Deng, X. Yang, T. Sellis, and J. X. Yu, "Most influential community search over large social networks," in *Proc. IEEE 33rd Int. Conf. Data Eng. (ICDE)*, Apr. 2017, pp. 871–882.
[33] R.-H. Li, L. Qin, F. Ye, J. X. Yu, X. Xiao, N. Xiao, and Z. Zheng, "Skyline community search in multi-valued networks," in *Proc. Int. Conf. Manage. Data*, Houston, TX, USA, May 2018, pp. 457–472.
[34] R.-H. Li, L. Qin, J. X. Yu, and R. Mao, "Influential community search in large networks," *Proc. VLDB Endowment*, vol. 8, no. 5, pp. 509–520, Jan. 2015.
[35] R.-H. Li, L. Qin, J. X. Yu, and R. Mao, "Finding influential communities in massive networks," *VLDB J.*, vol. 26, no. 6, pp. 751–776, Dec. 2017.
[36] Q. Liu, M. Zhao, X. Huang, J. Xu, and Y. Gao, "Truss-based community search over large directed graphs," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2020, pp. 2183–2197.
[37] Y. Liu, T. Safavi, A. Dighe, and D. Koutra, "Graph summarization methods and applications: A survey," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 62:1–62:34, Jun. 2018.
[38] G. Palla, I. Derényi, I. Farkas, and T. Vicsek, "Uncovering the overlapping community structure of complex networks in nature and society," *Nature*, vol. 435, no. 7043, pp. 814–818, Jun. 2005.
[39] A. E. Sarıyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek, "Incremental *k*-core decomposition: Algorithms and evaluation," *VLDB J.*, vol. 25, no. 3, pp. 425–447, Jun. 2016.
[40] A. E. Sariyüce, C. Seshadhri, and A. Pinar, "Local algorithms for hierarchical dense subgraph discovery," *Proc. VLDB Endowment*, vol. 12, no. 1, pp. 43–56, Sep. 2018.
[41] J. Shao, Z. Han, Q. Yang, and T. Zhou, *Community Detection Based on Distance Dynamics*. New York, NY, USA: ACM, 2015, pp. 1075–1084.
[42] M. Sozio and A. Gionis, "The community-search problem and how to plan a successful cocktail party," in *Proc. 16th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD)*, New York, NY, USA, 2010, pp. 939–948.
[43] C. Tsourakakis, F. Bonchi, A. Gionis, F. Gullo, and M. Tsiarli, "Denser than the densest subgraph: Extracting optimal quasi-cliques with quality guarantees," in *Proc. 19th Int. Conf. Knowl. Discovery Data Mining (ACM SIGKDD)*, New York, NY, USA, Aug. 2013, pp. 104–112.
[44] K. Varsha and K. K. Patil, "An overview of community detection algorithms in social networks," in *Proc. Int. Conf. Inventive Comput. Technol. (ICICT)*, Feb. 2020, pp. 121–126.
[45] Y. Wu, R. Jin, J. Li, and X. Zhang, "Robust local community detection: On free rider effect and its elimination," *Proc. VLDB Endowment*, vol. 8, no. 7, pp. 798–809, Feb. 2015.
[46] F. Zhang, Y. Zhang, L. Qin, W. Zhang, and X. Lin, "When engagement meets similarity: Efficient (k,r)-core computation on social networks," *Proc. VLDB Endowment*, vol. 10, no. 10, pp. 998–1009, Jun. 2017.
[47] Z. Zheng, F. Ye, R.-H. Li, G. Ling, and T. Jin, "Finding weighted k-truss communities in large networks," *Inf. Sci.*, vol. 417, pp. 344–360, Nov. 2017.
[48] Y. Zhou, H. Cheng, and J. X. Yu, "Graph clustering based on structural/attribute similarities," *Proc. VLDB Endowment*, vol. 2, no. 1, pp. 718–729, Aug. 2009.
[49] Q. Zhu, H. Hu, C. Xu, J. Xu, and W.-C. Lee, "Geo-social group queries with minimum acquaintance constraints," *VLDB J.*, vol. 26, no. 5, pp. 709–727, Oct. 2017.

**NARIMAN ADEL HUSSEIN** received the B.Sc. and M.Sc. degrees from the Department of Information Systems, Faculty of Computers and Artificial Intelligence, Cairo University, in 2005 and 2014, respectively, where she is currently pursuing the Ph.D. degree in information systems.

**HODA M. O. MOKHTAR** received the B.Sc. degree (Hons.) and the M.Sc. degree from the Department of Computer Engineering, Faculty of Engineering, Cairo University, in 1997 and 2000, respectively, and the Ph.D. degree in computer science from the University of California, Santa Barbara (UCSB), in 2005. She has taught multiple courses for the undergraduate and graduate levels and supervised a number of master's and Ph.D. theses with the Faculty of Computers and Artificial Intelligence, Cairo University, where she is currently the Chair of the Information Systems Department. She has participated in several national committees. Her research interests include big data analytics, data warehousing, data mining, database systems, social network analysis, bioinformatics, and web services. She received the Scholarship and the Dean's Fellowship from the Computer Science Department, UCSB, in 2000. She also received multiple awards and certificates for her academic achievements.

**MOHAMED E. EL-SHARKAWI** received the B.Sc. degree in systems and computer engineering from the Faculty of Engineering, Al-Azhar University, Cairo, Egypt, and the M.E. and Ph.D. degrees in computer science and communication engineering from the Faculty of Engineering, Kyushu University, Japan. He is currently a Professor with the Department of Information Systems, Faculty of Computers and Artificial Intelligence, Cairo University. His research interests include data engineering, including query processing, temporal databases, and social network analysis.