# Effective and Efficient Relational Community Detection and Search in Large Dynamic Heterogeneous Information Networks

Xun Jian
The Hong Kong University of Science and Technology
Hong Kong, China
xjian@cse.ust.hk

Yue Wang[*]
Shenzhen Institute of Computing Sciences, Shenzhen University
Shenzhen, China
yuewang@sics.ac.cn

Lei Chen
The Hong Kong University of Science and Technology
Hong Kong, China
leichen@cse.ust.hk

## ABSTRACT

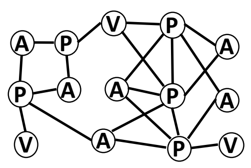
Community search in heterogeneous information networks (HINs) has attracted much attention in graph analysis. Given a vertex, the goal is to find a densely-connected subgraph that contains the vertex. In practice, the user may need to restrict the number of connections between vertices, but none of the existing methods can handle such queries. In this paper, we propose the relational constraint that allows the user to specify fine-grained connection requirements between vertices. Base on this, we define the relational community as well as the problems of detecting and searching relational communities, respectively. For the detection problem, we propose an efficient solution that has near-linear time complexity. For the searching problem, although it is shown to be NP-hard and even hard-to-approximate, we devise two efficient approximate solutions. We further design the round index to accelerate the searching algorithm and show that it can handle dynamic graphs by its nature. Extensive experiments on both synthetic and real-world graphs are conducted to evaluate both the effectiveness and efficiency of our proposed methods.

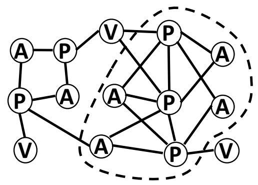
**PVLDB Reference Format:**
Xun Jian, Yue Wang, and Lei Chen. Effective and Efficient Relational Community Detection and Search in Large Dynamic Heterogeneous Information Networks. *PVLDB*, 13(10): 1723-1736, 2020.
DOI: https://doi.org/10.14778/3401960.3401969

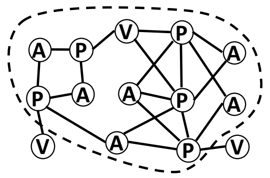
## 1. INTRODUCTION

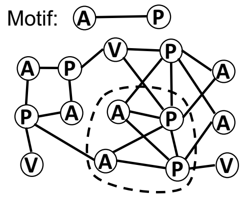
### 1.1 Motivations

In real-world applications, we often model the underlying data as graphs. For example, the World Wide Web can be treated as a graph, where each web page is represented by a vertex, and hyperlinks between pages are represented by edges between vertices. Such a graph is also known as a *homogeneous network*, since all vertices in this graph have the same type (or label). Another kind of graph, in which vertices have different types, are called *heterogeneous information networks* (HINs). For example, Figure 1a shows an HIN representing a bibliographical network. In this network, vertices with labels A, P and V represent authors, papers and venues, respectively. Edges connecting different types of vertices have different semantics such as *Authorship* (author-paper) and *Citation* (paper-paper). Compared to the homogeneous network, the HIN stores rich semantic information and has attracted much attention recently in research areas including search [32,35], clustering [17,36,37] and data mining [31,33].


(a) A bibliographical network G.


(b) The desired community in G.


(c) A 2-core in G.



(d) A maximal m-Clique in G.

**Figure 1: An example bibliographical network.**

It is shown that many real-world networks have a significant property of community structure [14], where vertices within a community are densely connected. Due to its wide applications [6, 19, 29, 38], retrieving communities becomes an important task in graph analysis, and has attracted much attention in the literature.

In HINs, works like [11,17,20,25] have been proposed to query communities by adopting traditional models such as $k$-core, $k$-truss and $k$-clique. However, there are still some

[*]Yue Wang is the corresponding author.

*Proceedings of the VLDB Endowment,* Vol. 13, No. 10
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3401960.3401969

users' needs that cannot be handled by existing methods. Sometimes, users want to find a community in which every vertex of type A has at least $k$ neighbors of type B. In fact, they may have different requirements for different types of vertices. Here we describe two typical situations in the following examples.

EXAMPLE 1. *Alice is studying the bibliographical network in Figure 1a. To identify active research groups, she would like to find some authors who frequently publish papers together. Specifically, she wants to find a community of authors and papers, such that each author publishes at least* 2 *papers in the community, and each paper is co-authored by at least* 3 *authors in the community. A possible community that she wants is shown in Figure 1b. However, existing models cannot accomplish this task very well. For example, if using $k$-cores, she cannot set degree requirements for authors and papers separately with a single parameter $k$. When setting $k = 2$, the 2-core includes all authors and papers (Figure 1c). When setting $k = 3$, then no results can be found. If using m-Cliques [17], she may miss some interesting results because m-Clique is too restricted. In Figure 1d, even with the simplest motif, she can only get part of the desired result.*

EXAMPLE 2. *Bob is an entrepreneur who wants to form a startup team by searching a human resource network, in which each person can be a manager or an employee. In order to help his team members get to know each other quickly, he requires that each employee must have contacts with another one. Similarly, he wants the manager to have contacts with at least 3 employees. In addition, to save money, he wants the team size to be minimal. Apparently, existing models such as k-core, k-truss and m-Clique cannot handle these requirements precisely, since they cannot handle specific constraints between different types of vertices. So, how can Bob query a desired team in the network?*

From the above examples, it is practical and important to investigate methods for community detection/searching, which can handle those fine-grained requirements.

## 1.2 Contributions

In this paper, we propose the *relational community* (r-com) in HINs that considers vertex degree more precisely. Instead of using a single parameter $k$ of $k$-core, we use a set of relational constraints, where each constraint describes "every vertex in type $T_a$ must have at least $k$ neighbors in type $T_b$". By combining several constraints, we can control the minimum degree between any pair of vertex types. Figure 2 shows 2 relational constraints, where $T_a$ is shaded, and $T_b$ is unshaded. In summary, it says "each author published $\geq$ 2 papers, and each paper has $\geq$ 3 authors". In an HIN, we can find a subgraph, where every vertex satisfies these relational constraints. We call such a subgraph a *relational community*. In Figure 1, the maximum r-com in $G$ that satisfies these two constraints is exactly the desired community of Alice.

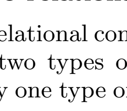


**Figure 2: Two relational constraints.**

The number in each relational constraint controls the minimum degree between two types of vertices. As one constraint is applied to only one type of vertices, we can specify flexible and precise requirements on every type of vertices by combining multiple constraints. For example, in a bibliographical network, we can set a large number on the paper-paper relation, to require papers in the community densely citing each other. Meanwhile, we can set a smaller number on the author-paper relation, to ensure each author publishes a reasonable (not too many) number of papers.

In this paper, we study the problems of decomposing and searching r-coms in HINs, which are referred to as *relational community detection* (RCD), and *minimum relational community search* (MRCS), respectively. The RCD problem is to find all maximal r-coms in an HIN. The output is useful when the user wants to understand the properties of the graph in a macro view. On the other hand, some users may only care about a particular part of the graph around a query vertex. Also, as Example 2 shows, they may require the community to be small so as to fit their actual needs. Based on this, we define the MRCS problem as to find the smallest r-com containing a specific vertex.

By investigating these two problems, we show that, RCD can be solved in polynomial time (PTIME). We further design a message-passing algorithm that runs in near-linear time, which is close to optimal since we have to traverse the whole graph. Nevertheless, MRCS is NP-hard and hard-to-approximate. Despite its hardness, we propose one exact algorithm and two heuristics to solve this problem. Specifically, we propose a greedy algorithm as well as a local search approach, and design a novel *round index* to accelerate the local search. Besides its efficiency, this index can handle graph changes naturally, so we can also use it to maintain an r-com when the graph is dynamically changing. This is important because real-world networks are known to be highly dynamic.

In summary, we make the following contributions.

- We propose a new community structure and formulate its detection (RCD) and searching (MRCS) problems;
- We develop an efficient algorithm that runs in near-linear time for RCD in static HINs;
- Though MRCS is NP-hard and hard-to-approximate, we develop exact and approximate algorithms for it;
- We design the round index to accelerate the approximate algorithm for MRCS, and show that this index can efficiently handle dynamic graphs;
- We conduct extensive experiments on large synthetic and real-world HINs to evaluate (1) the effectiveness of r-com in HIN analysis, (2) the efficiency of our proposed algorithms for RCD and MRCS, and (3) the efficiency of proposed methods for MRCS on dynamic graphs. The results show that r-com well captures the type information in HINs, and our methods can efficiently solve the two problems.

The rest of this paper is organized as follows. In Section 2, we define the target problems and then investigate their hardness. In Section 3 and Section 4, we propose several algorithms to solve the RCD and MRCS problems, respectively. An incremental algorithm is also devised to handle dynamic graphs. In Section 5, we conduct extensive experiments on both synthetic and real-world HINs, to show the effectiveness and efficiency of both our proposed community structure and the solutions. Section 8 surveys the related works and compares them to our work. Finally, we conclude in Section 7.

## 2. PROBLEM DEFINITION

In this section, we first introduce some background terms and definitions. Then we formally define the problems of detecting and searching relational communities, and discuss their hardness.

## 2.1 Problem Definition

In this work, we model an HIN as an undirected graph $\mathrm{G}(\mathrm{V_G}, \mathrm{E_G}, \mathrm{L_G}, \phi)$, where $\mathrm{V_G}$ is the vertex set, $\mathrm{E_G}$ is the edge set, and $\mathrm{L_G}$ is the label set. Function $\phi : \mathrm{V_G} \to \mathrm{L_G}$ assigns each vertex $v$ a label $\phi(v)$. For any $H \subseteq \mathrm{V_G}$, its *induced subgraph* in G, denoted as $\mathrm{G}[H]$, is a graph who has vertex set $H$ and edge set $(H \times H) \cap \mathrm{E_G}$. For vertex $v$, we define its neighbors as its adjacent vertices, i.e. $\mathcal{N}_\mathrm{G}(v) = \{u|(v,u) \in \mathrm{E_G}\}$, and its degree $d_v = |\mathcal{N}_\mathrm{G}(v)|$. More precisely, $\mathcal{N}_\mathrm{G}(v,l)$ denotes the neighbors of $v$ that have label $l$, i.e. $\mathcal{N}_\mathrm{G}(v,l) = \{u|(v,u) \in \mathrm{E_G}, \phi(u) = l\}$.

In this work, we use relational constraints to test whether a vertex belongs to a community. Formally, a constraint $s$ is a triplet $\langle l_1, l_2, k\rangle$, where $l_1, l_2 \in \mathrm{L_G}$, and $k \geq 1$. It means "each vertex with label $l_1$ must have at least $k$ neighbors with label $l_2$". In other words, given any graph $G'$, vertex $v$ *satisfies* $s = \langle l_1, l_2, k\rangle$ if either of the conditions below holds:

1. $\phi(v) \neq l_1$, or
2. $\mathcal{N}_{G'}(v, l_2) \geq k$.

Here condition 1 means the constraint $s$ is not applicable to $v$, and for simplicity we also say $v$ satisfies $s$ in this situation.

Remark 1. *We focus on HINs where edges have no labels or directions in this work. Our techniques can be easily extended on directed and/or labeled edges by only changing the relational constraints. For example, we can define the constraint $s = \langle l_1, l_2, l_3, k\rangle$, which means "for each $v$ with label $l_1$, there must be at least $k$ vertices with label $l_2$ that link to $v$ by an edge with label $l_3$". In this way, we can handle edges with labels.*

A user can use a set of constraints $S = \{s_1, s_2, \dots, s_t\}$ to specify composite requirements for different types of vertices. It controls what a community looks like, and is thus called the *community schema*. Together with $S$, the user also implicitly defines a *concerned label set* $L_S$ that contains all labels appeared in $S$, namely

$$L_S = \{l|\langle l, l', k\rangle \in S \text{ or } \langle l', l, k\rangle \in S\}.$$

Labels that do not appear in any constraint are of no interest to the user, so vertices with these labels can be excluded automatically. This is important because a user may only care about a few labels, while HINs like knowledge graphs can contain hundreds or even thousands of them. In summary, given a graph $G$ and a schema $S$, we say a vertex $v$ is *qualified* if (1) $\phi(v) \in L_S$ and (2) $v$ satisfies all constraints in $S$. Otherwise, we say $v$ is *unqualified*. Then we say $G$ is a *relational community* (r-com) if every vertex is qualified with respect to $S$.

Definition 1 (Relational Community). *Given $S$ as well as $L_S$, a connected graph $R$ is a relational community (r-com) defined by $S$ if and only if $\forall v \in V_R$, (1) $\phi(v) \in L_S$; and (2) $v$ satisfies all constraints in $S$. When the context is clear, we simply say $R$ is a relational community.*

Though a graph itself may not be an r-com, some of its subgraphs might be. So detecting all maximal r-coms in a graph is a practical and interesting problem.

Definition 2 (RCD). *Given G, $L_S$ and $S$, to find all subsets $H_i \subseteq \mathrm{V_G}$, such that*

1. *$\mathrm{G}[H_i]$ is a relational community, and*
2. *$H_i$ is maximal, i.e., $\forall V' \subseteq \mathrm{V_G}$ and $V' \cap H_i \neq V'$, $\mathrm{G}[H_i \cup V']$ is not a relational community.*

On the other hand, in some applications, a user only wants a small community in a specific area in the graph. We thus propose the MRCS problem below.

Definition 3 (MRCS). *Given G, $L_S$, $S$ and a query vertex $q$, to find $H \subseteq \mathrm{V_G}$, such that*

1. *$q \in H$, and*
2. *$\mathrm{G}[H]$ is a relational community, and*
3. *$|H|$ is minimized.*

**Discussion.** The $k$-core is a specialization of the r-com in homogeneous networks. A homogeneous network can be treated as an HIN where all vertices have the same type, say $l_0$. In this case, the user can only specify one relational constraint in the schema, which is $\langle l_0, l_0, k\rangle$. It requires that every vertex in the r-com has at least $k$ neighbors, so such an r-com becomes exactly a $k$-core.

## 2.2 Problem Hardness

We now discuss the hardness of RCD and MRCS. In fact, RCD can be solved in polynomial time. A simple solution would be to gradually remove vertices that do not satisfy some constraints in $S$. In section 3, we will first analyze the correctness of this algorithm, and then propose a more efficient method.

On the other hand, MRCS is NP-hard and even hard-to-approximate, so it is unlikely to solve this problem accurately in polynomial time. The decision version of MRCS ($\mathrm{MRCS_D}$) is to determine whether a relational community $\mathrm{G}[H]$ exists, where $q \in H$ and $|H| = m$. We prove it to be NP-complete by reducing from an existing NP-complete problem, Maximum Clique (MC) [23]. Its decision version ($\mathrm{MC_D}$) is to determine if a graph contains a clique of size $k$.

Lemma 1. *$MRCS_D$ is NP-complete, and thus MRCS is NP-hard.*

Proof. *It is easy to see that $MRCS_D$ is in NP, because we can check whether $\mathrm{G}[H]$ is a relational community in polynomial time. Now we show it is also NP-hard by reducing $MC_D$ to it. Given an unlabeled graph $G(V,E)$ and $k$, $MC_D$ is to determine if we can find an $H \subseteq V$ such that $G[H]$ is a $k$-clique. We can build another labeled graph $G'(V',E')$ by adding a new vertex $u$ that connects all vertices in $V$. In addition, we assign label $a$ to every vertex in $V$, and label $b$ to $u$. Now we create an instance of $MRCS_D$, where $C = \{\langle b, a, k\rangle, \langle a, a, k-1\rangle\}$, $q = u$, and $m = k+1$. That is, we want to find $k+1$ vertices (include $u$) in $V'$, and they are fully connected to each other. Apparently, a feasible solution to this problem corresponds to a $k$-clique in $G$.* $\square$

We further prove that MRCS is hard-to-approximate, i.e., there is no polynomial-time algorithm for MRCS with a constant approximation factor unless P=NP.

Lemma 2. *There is no polynomial-time algorithm for MRCS with a constant approximation factor, unless P=NP.*

Proof. *We prove this by contradiction. Suppose that there exists a polynomial-time algorithm $\Pi_1$ for MRCS with an approximation factor $\rho$. Then given graph G, schema $S$,* *vertex $u$ and $\Pi_1$'s output $R$, we have $|V_R| \leq \rho \cdot |V_{O_u}|$, where $O_u$ is the optimal solution. We now show that using $\Pi_1$ we can solve the Minimum Subgraph of Minimum Degree $\geq d$ ($MSMD_d$) problem [2] with approximation factor $\rho$.*

*Given a graph $G$ and an integer $d$, the $MSMD_d$ problem is to find a d-core $D$, such that $|V_D|$ is minimized. Now we design an algorithm $\Pi_2$ that uses $\Pi_1$ to solve $MSMD_d$ in four steps:*

1. *Assign every vertex in $G$ with label $l_0$;*
2. *Create a schema $S = \{\langle l_0, l_0, d\rangle\}$;*
3. *For each vertex $u$ in $V_G$, use $\Pi_1$ to find the minimum r-com $R_u$ that contains $u$;*
4. *Output $R = \underset{R_u \neq \emptyset}{\operatorname{argmin}} |V_{R_u}|$.*

*It is easy to verify that $\Pi_2$ runs in polynomial time. Besides, each r-com defined by $S$ is a d-core, and vise versa. Thus, the final output $R$ is a feasible solution of $MSMD_d$. Let $OPT$ be the optimal solution of $MSMD_d$, and $u$ be a vertex in $V_{OPT}$, then $OPT$ is also the optimal solution of MRCS given $u$ (i.e., $O_u$). According to our assumption, we have $|V_R| \leq |V_{R_u}| \leq \rho \cdot |V_{O_u}| = \rho \cdot |V_{OPT}|$. That is, $\Pi_2$ solves $MSMD_d$ with approximation factor $\rho$. However, it is proved that $MSMD_d$ is hard-to-approximate unless P=NP [2], so our assumption is incorrect, and $\Pi_1$ does not exist.* □

### 2.3 Community Schema Discovering

As Example 1 and Example 2 show, when the community structure is clear, a user can easily construct the schema $S$. However, it might be hard to choose what constraints to use if the user is looking for a good schema through trial-and-error. To help the user overcome such hardness, we propose to reverse-engineering a high-quality schema $S$ from an exemplary community $R$.

Specifically, for each pair of labels $l_1, l_2 \in L_R$, we put a constraint $\langle l_1, l_2, k\rangle$ in $S$, where $k = \min_{v \in V_R, \phi(V)=l_1} |\mathcal{N}_R(v, l_2)|$. That is, while keeping every vertex in $V_R$ satisfying all constraints in $S$, we put as many as constraints in $S$ and set the largest possible value for each of them. We expect that, with only a few modifications, this relation can be used to find communities that are similar to $R$. Intuitively, the result schema $S$ is a promising starting point for detecting/searching good communities, because it is built from a known community. In practice, we will illustrate how we use this method to effectively detect communities from real-world networks in section 5.

## 3. SOLVING THE RCD PROBLEM

In this section, we devise efficient solutions for the RCD problem. Specifically, we first design a naive solution, which runs in quadratic time with respect to the graph size. Then we propose a non-trivial message-passing approach that runs in near-linear time when the schema isn't too large.

### 3.1 The Naive Solution

A simple idea to solve the RCD problem would be gradually removing vertices that are not qualified. After removing those vertices, some originally qualified ones may become unqualified due to the loosing of neighbors. By repeatedly removing them, we finally get a graph in which all vertices are qualified, and each connected component is a maximal r-com. We summarize this algorithm in algorithm 1. It iteratively identifies unqualified vertices (lines 3-5), and removes them from $G$ (line 6). When $H = \emptyset$, all rest vertices in $G$ are qualified, so it returns each connected component in $G$ as a maximal r-com. The correctness of algorithm 1

**Algorithm 1:** RCD Naive

```
Input  : $G$, $S$.
Output: a set of maximal r-coms.
1 do
2     $H \leftarrow \emptyset$;
3     foreach $v \in V_G$ do
4         if $v$ is not qualified then $H \leftarrow H \cup \{v\}$ ;
5     $V_G \leftarrow V_G \setminus H$;
6 while $H \neq \emptyset$;
7 return all connected components in $G$;
```

can be justified by Proposition 1. Basically, if a vertex is unqualified in $G$, it cannot be qualified in any of $G$'s subgraphs. This means we can remove $v$ safely. It follows that maximal r-coms have no overlaps with each other.

PROPOSITION 1. *Given $S$, $G$, $V' \subseteq V_G$, and $v \in V'$, if $v$ is unqualified in $G$, then $v$ is also unqualified in $G[V']$.*

PROOF. *There are two cases if $v$ is unqualified in $G$.*

*(1) $\phi(v) \notin L_S$. This is a trivial case.*

*(2) $v$ does not satisfy a constraint $\langle \phi(v), l, k\rangle \in S$, i.e., $\mathcal{N}_G(v, l) < k$. In this case, we have $\mathcal{N}_{G[V']}(v, l) < k$ since $\mathcal{N}_{G[V']}(v) \subseteq \mathcal{N}_G(v)$.* □

COROLLARY 1. *Given a maximal r-com $G[H]$ and an arbitrary r-com $G[H']$, either $H' \subset H$, or $H \cap H' = \emptyset$.*

PROOF. *If $H' \subset H$, it is trivial. Otherwise, since $H$ is maximal, $G[H \cup H']$ is not an r-com. Therefore there is at least one vertex $v \in H \cup H'$, such that $v$ is not qualified in $G[H \cup H']$. Suppose $v \in H$, then follow Proposition 1 we know $v$ is not qualified in $G[H]$. This contradicts with the input that $G[H]$ is an r-com. The same applies if $v \in H'$.* □

It is obvious that algorithm 1 runs in polynomial time. In each round, at least one vertex is removed from $G$, so there are at most $|V_G|$ rounds. In a round, we enumerate all vertices in $V_G$, to check whether each of them is qualified (line 4). Line 4 can be done by scanning the neighbors of $v$ once and counting the numbers of each label, and then check each constraint in $S$ one by one. So in summary, lines 3-5 scan each edge exactly twice, and scan $S$ for $|V_G|$ times, which takes $O(|E_G| + |S| \cdot |V_G|)$ time. In line 10 we can use *breadth-first search* to find connected components in $G$ in $O(|E_G|)$ time [16], so the total running time of algorithm 1 is $O(|V_G| \cdot |E_G| + |S| \cdot |V_G|^2)$.

### 3.2 A Message-Passing Approach

Though the naive algorithm runs in polynomial time, it can be slow when the graph becomes large. The main drawback is it always checks every vertex in every round, even there is no necessity. As an example, consider a qualified vertex $v$ in the $i$th round. If none of $v$'s neighbors is removed in $i$th round, we immediately know that $v$ will still be a qualified vertex in the $(i+1)$th round because $\mathcal{N}_G(v)$ does not change. In this case, we can pass $v$ when enumerating and checking vertices, and thus save the running time. On the other hand, if one of $v$'s neighbor $u$ is removed, we know that $v$ may become unqualified in the next round and thus needs to be checked.

To avoid unnecessary computations, we devise a message-passing algorithm to remove unqualified vertices in a check-when-necessary manner. Specifically, we first scan and check every vertex for once. In this pass, unqualified vertices are removed, and each of them sends a message for each of its neighbors. A message has the form $(v, u)$, which means "$v$'s neighbor $u$ is removed". Then we check whether vertex $v$ remains to be qualified when a message $(v, u)$ is received.

We summarize this algorithm in algorithm 2. In lines 3-11, we scan all vertices once, remove unqualified vertices, and push associated messages into a queue. In lines 12-18, we handle messages in the queue one by one. For message $(v, u)$, we check if it is still in $V_G$ and becomes unqualified (line 14). If so, we push a message in to the queue for each of its neighbors, and remove it from $V_G$. The correctness of algorithm 2 can be proved by comparing the vertex removals to those in algorithm 1. Consider a vertex $v$ which is removed in the 1st round in algorithm 1. In algorithm 2 it will be removed after the first for-loop (line 11). Then we look at vertex $v'$ which is removed in the 2nd round in algorithm 1, which becomes unqualified after a set $N \subseteq \mathcal{N}_G(v')$ of its neighbors are removed in the 1st round. We have shown that all vertices in $N$ will be removed in algorithm 2, so $|N|$ messages are pushed into the message queue for checking $v'$. After processing the last message, $v$ will be found unqualified and removed. Thus, all vertices removed in the 2nd round in algorithm 1 will also be removed in algorithm 2. Following this way, we can prove that all vertices removed in algorithm 1 will be removed in algorithm 2. On the other hand, algorithm 2 does not remove qualified vertices, which completes our proof.

**Algorithm 2:** Message Passing

```
Input  : $G$, $S$.
Output: a set of maximal r-coms.
1  Queue ← $\emptyset$, $H \leftarrow \emptyset$;
2  foreach $v \in V_G$ do
3      if $v$ is not qualified then
4          foreach $u \in \mathcal{N}_G(v)$ do Queue.push($(u, v)$) ;
5          $H \leftarrow H \cup \{v\}$;
6  $V_G \leftarrow V_G \setminus H$;
7  while Queue $\neq \emptyset$ do
8      $(v, u)$ ← Queue.pop();
9      if $v \in V_G$ and $v$ is not qualified then
10         foreach $n \in \mathcal{N}_G(v)$ do Queue.push($(n, v)$) ;
11         $V_G \leftarrow V_G \setminus \{v\}$;
12 return all connected components in $G$;
```

The advantage of message-passing is eliminating part of redundant calculations. It checks zero or one vertex for each message pushed into the message queue, hence the total number of checked vertices is no larger than the total number of pushed messages. Meanwhile, the number of pushed messages is bounded by $2 \cdot |E_G|$, because for each edge, the removal of its either side produces one message. Therefore, if checking a vertex takes $O(|S| + d_{max})$ time, the total running time of algorithm 2 is $O((|E_G| + |V_G|) \cdot (|S| + d_{max}))$, where $d_{max}$ is the maximum degree of vertices.

**Counting Index.** Considering that $|L_S|$ might be small, we can further reduce the running time using a counting index for every vertex. The intuition is that we can use messages as records of graph changes, to avoid scanning neighbors of each vertex repeatedly. Specifically, we maintain a table $T$ where $T_{v,l}$ stores $|\mathcal{N}_G(v, l)|$. This table can be constructed when we checking each vertex during the first for-loop (line 4). Later when handling each message $(v, u)$ (line 14), we can simply deduct $T_{v,\phi(u)}$ by 1, and then check the constraint $\langle \phi(v), \phi(u), k \rangle$. The total running time hence reduces to $O(|V_G| \cdot |S| + |E_G|)$, which is near-linear time if $|S|$ is smaller than the average degree of $G$. On the other hand, we only need to store counts for labels in $L_S$, so the space complexity is $O(|V_G| \cdot |L_S|)$.

## 4. SOLVING THE MRCS PROBLEM

As MRCS is NP-hard, it is unlikely that we can solve it both accurately and efficiently. In this section, we first design an exact algorithm which has exponential running time, then we propose two approximation polynomial-time algorithms, Greedy and Local Search.

### 4.1 An Exact Solution

Recall that the MRCS problem is to find the minimum r-com that contains $q$, denoted as $R_{min}(q)$. A simple idea would be enumerating all subsets $H \subseteq V_G$ containing $q$, and pick the smallest one such that $G[H]$ is an r-com. However, in many cases $G[H]$ is not an r-com, or even not connected. To avoid encountering such trivial situations, we design an algorithm to enumerate $H$ in a vertex-removal manner.

Specifically, we start from the maximum r-com that contains $q$, denoted as $R_{max}(q)$. From Corollary 1 we know that $R_{min}(q)$ is a subgraph of $R_{max}(q)$. Thus, we can gradually remove vertices in $R_{max}(q)$, until we get a minimal graph $R'$ which is an r-com containing $q$. Then for all possible $R'$s, the one having minimum vertices would be $R_{min}(q)$. During this procedure, the key is to check whether a candidate $R'$ is an r-com and whether it is minimal. To facilitate our analysis, we define the *vertex group* as follows.

Definition 4 (Vertex Group). *Given schema $S$, an r-com $R$, and $v \in V_R$, the vertex group of $v$, denoted as $VG(R, v)$, is the minimum set $V' \subseteq V_R$, such that (1) $v \in V'$, and (2) $R[V_R \setminus V']$ is either an empty graph or an r-com.*

In other words, if we want to remove $v$ from $R$, we must remove all vertices in $VG(R, v)$ as well, otherwise, there will be unqualified vertices left in the graph. To get $VG(R, v)$ for a specific vertex $v$, we can first remove $v$ from $R$, and then use the message-passing algorithm for the RCD problem to find reset vertices.

Instead of removing vertices one by one from $R_{max}(q)$, now we can gradually remove vertex groups, which reduces the search space, and Proposition 1 guarantees that we can get the correct result. Besides, we cannot remove a vertex group that contains $q$, so when all vertex groups contain $q$, we know that the current r-com is minimal.

Proposition 2. *An r-com $R$ is the minimal one that contains $q$, if and only if $\forall v \in V_R$, $q \in VG(R, v)$.*

We summarize the above search algorithm in algorithm 3. In line 1 it invokes the subroutine MessagePassing (algorithm 2) to find all maximal r-coms in $G$. Then in line 2 it picks the maximal r-com that contains $q$, which is $R_{max}(q)$. Finally in line 3 it invokes another subroutine FindMinimum (algorithm 4) to find $R_{min}(q)$ from $R_{max}(q)$.

Algorithm 4 searches for the $R_{min}(q)$ in a recursive way. It first collects all vertex groups that can be removed (i.e., do

**Algorithm 3:** MRCS Exact

```
Input  : $G$, $S$, $q$.
Output: $R_{min}(q)$.
1 $\mathcal{R} \leftarrow$ MessagePassing ($G$, $S$);
2 $R_{max}(q) \leftarrow$ the r-com in $\mathcal{R}$ that contains $q$;
3 $R_{min}(q) \leftarrow$ FindMinimum ($R_{max}(q)$, $S$, $q$);
4 return $R_{min}(q)$;
```

not contain $q$) in line 1. If no vertex group can be removed, the current $R$ is the minimal one, so the algorithm returns $R$ as the result. Otherwise in line 5, it tries to remove each vertex group, and invokes itself recursively to find the minimum r-com in the rest vertices (here we use $|\cdot|$ to denote the number of vertices in the returned r-com). Among all choices, it picks the r-com with minimum vertices, and returns as its result. This algorithm ends for sure, because for each recursive call, the size of $V_R$ strictly reduces, and the size of $\mathcal{D}$ is bounded by $V_R$. In fact, in the worst case $\forall v \in V_R$, $VG(R, v) = \{v\}$, then algorithm 4 needs to try removing all combinations of vertex groups, the number of which is $2^{|V_R|}$. For each combination, identifying and removing all vertex groups take in total $O(|E_R|)$ time, as they are linear-time algorithms. Thus the running time of this algorithm is $O(2^{|V_R|} \cdot |E_R|)$. Its correctness is proved by the discussion above.

**Algorithm 4:** Find Minimum

```
Input  : $R$, $S$, $q$.
Output: the minimum r-com in $R$ that contains $q$.
1 $\mathcal{D} \leftarrow \{VG(R,v) | \forall v \in V_R, q \notin VG(R,v)\}$;
2 if $\mathcal{D} = \emptyset$ then return $R$ ;
3 $H' \leftarrow \text{argmin}_{H \in \mathcal{D}}$ |FindMinimum($R[V_R \setminus H]$, $S$, $q$)|;
4 return FindMinimum($R[V_R \setminus H']$, $S$, $q$);
```

## 4.2 A Greedy Approach

The main drawback of the exact solution is its exponential complexity. When the graph is large, it is not practical to wait for an exact result. Instead, we may look for approximate results that can be obtained in a reasonable time.

Here we propose a greedy algorithm, which is a simple modification of the exact solution. The intuition is that, other than globally minimizing the size of $R$, only picking the local minimum at each step may still lead to a good solution. Since we do not need to try every choice at each step, the running time would be reduced significantly.

We summarize this approach in algorithm 5. In lines 1-2, it gets $R_{max}(q)$, and in lines 3-9 it iteratively removes the largest vertex group from the r-com. When there is no vertex group can be removed, a minimal r-com is found and returned as the result.

As for the time complexity, lines 1-2 take $O(|E_G|)$ time as discussed before. In each iteration, line 4 takes $O(|E_R| \cdot |V_R|)$ time to get the vertex group of every vertex in $V_R$, and then lines 6-7 take no more than $O(|E_R|)$ time to remove $H'$. The maximum number of iterations is $|V_{R_{max}(q)}|$, given that in each iteration at least one vertex is removed. Therefore, the total running time is $O(|E_G| + |E_{R_{max}(q)}| \cdot |V_{R_{max}(q)}|^2)$.

## 4.3 A Local Search Approach

Considering that in an r-com $R$ which is a connected graph, $|E_R| \geq |V_R| - 1$, the greedy approach has a time

**Algorithm 5:** MRCS Greedy

```
Input  : $G$, $S$, $q$.
Output: $R_{min}(q)$.
1 $\mathcal{R} \leftarrow$ MessagePassing ($G$, $S$);
2 $R \leftarrow$ the r-com in $\mathcal{R}$ that contains $q$;
3 do
4     $\mathcal{D} \leftarrow \{VG(R,v) | \forall v \in V_R, q \notin VG(R,v)\}$;
5     if $\mathcal{D} \neq \emptyset$ then
6         $H' \leftarrow \text{argmax}_{H \in \mathcal{D}} |H|$;
7         $R \leftarrow R[V_R \setminus H']$
8 while $\mathcal{D} \neq \emptyset$;
9 return $R$;
```

complexity which is at least cubic to $|V_{R_{max}(q)}|$. This would be an issue when $|V_{R_{max}(q)}|$ is large. Another issue is that it needs to traverse the whole graph $G$ to find $R_{max}(q)$, which is slow given that graphs are large in real applications.

To counter these two issues, we propose a local-search approach that only reads the necessary part of the whole graph. Basically, we start from a vertex set $Q = \{q\}$, and gradually add vertices into $Q$, until an r-com $R$ containing $q$ appears in $G[Q]$. Then we report a minimal r-com inside $R$ as the result. Intuitively, if a feasible solution happens to be in a small area around the query vertex $q$, such an approach can quickly find it. Meanwhile, it only reads vertices in $Q$ and their adjacent edges, so it can be applied even if the graph cannot fit in the main memory.

**Candidate Selection.** When we gradually adding vertices into $Q$, the order we adding them decides how quickly we can find an answer. In this paper, we maintain a candidate set, in which each vertex is a neighbor of vertices in $Q$, and always select the vertex with the largest priority as the next one to be added. The priority of vertex $v$ is defined as

$$pri(v) = \frac{|\mathcal{N}_G(v) \cap Q|}{dist(v)}$$

, where $dist(v)$ is the distance of $v$ to $q$. This priority function balances the search depth (the less the more important) and vertex connectivity (the more the more important). The intuition is that we want to find vertices that are close to $q$ and densely-connected as well. After adding a vertex into $Q$, we also add all its neighbors into the candidate set.

**Candidate Pruning.** A vertex should not be picked as a candidate if it cannot be in any r-com. For example, by Proposition 1, if a vertex is unqualified in $G$, it is not a candidate. Besides, when those vertices are identified, we may find more vertices that cannot be in any r-com. A simple example is shown in Figure 3, in which $v_5$ is unqualified and $v_4$ "looks" qualified. Since $v_5$ cannot be in any r-com, it can be treated as not existed, and then we can identify that $v_4$ cannot be in any r-com, neither. To prune such vertices, we maintain the counting index (Section 3.2) for each candidate. Whenever we identify a non-candidate, we deduct the counting index for each of its neighbors in our candidate set. Then more non-candidate vertices can be identified by the updated counting index, and so on so forth.

In this approach, a key operation is to identify when such an $R$ appears while expanding $Q$. We can simply use algorithm 1 or algorithm 2 to find $R$ whenever adding a new vertex, but the time complexity would be at least $O(|Q|^2)$, which is not efficient. In this work, we propose a *round in-*

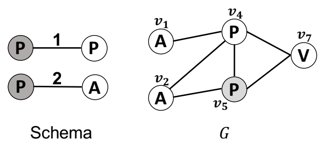


**Figure 3: A candidate pruning example.**

*dex* to "monitor" all r-coms when the graph changes, and consequently identifies the appearing of $R$. It is expected that when a new vertex is added into $Q$, we only need to update part of the index by reading a few edges in $G[Q]$ and thus can improve the efficiency.

## 4.4 The Round Index

When using algorithm 1 to detect r-coms in $G[Q]$, it removes vertices round by round. In this part, we design an efficient round index to track vertex removal, and to avoid running algorithm 1 repeatedly.

The round index is designed as a round number $r_v$ ($r_v \geq 1$) associated with each vertex $v$, indicating that $v$ is removed in the $r_v$-th round. As a special case, for vertices that are never removed (i.e., belong to an r-com), we set $r_v = \infty$. Given graph $G' = G[Q]$ and schema $S$, we say a round index is *correct* if every vertex $v \in V_{G'}$ is truly removed in the $r_v$-th round after running algorithm 1. We prove in Lemma 3 and Lemma 4 that this correctness can be verified by the following equation:

$$r_v = \underset{i \in [1,\infty]}{\operatorname{argmin}}(v \text{ is unqualified with respect to } \mathcal{N}^i_{G'}(v)) \quad (1)$$

, where $\mathcal{N}^i_{G'}(v)$ is the neighbor set of $v$ at round $i$:

$$\mathcal{N}^i_{G'}(v) = \begin{cases} \{u | u \in \mathcal{N}_{G'}(v), r_u \geq i\} & \text{, if } i \neq \infty; \\ \emptyset & \text{, if } i = \infty. \end{cases}$$

LEMMA 3. *Given graph $G'$ and schema $S$, vertex $v$ belongs to an r-com **if and only if** Equation 1 holds for every vertex and $r_v = \infty$.*

PROOF. *(if) In Equation 1, for a sufficiently large (but finite) $i$, $\mathcal{N}^i_{G'}(v) = \{u | u \in \mathcal{N}_{G'}(v), r_u = \infty\}$. If $r_v = \infty$, then $v$ is qualified with respect to $\mathcal{N}^i_{G'}(v)$. Let $H = \{u | u \in V_{G'}, r_u = \infty\}$, and $A = G'[H]$, then $\mathcal{N}_A(v) = \mathcal{N}^i_{G'}(v)$, and thus $v$ is qualified in $A$. It follows that every vertex in $A$ is qualified, and $v$ belongs to an r-com in $A$.*

*(only if) We prove it by contradiction. Suppose $v$ belongs to an r-com $R = G'[H]$, and $r_v \neq \infty$. Without loss of generality, we assume that $r_v$ is the smallest among all vertices in $H$. Let $i = \min_{u \in \mathcal{N}_R(v)} r_u$. It is obvious that $\mathcal{N}^i_{G'}(v) = \mathcal{N}_R(v)$. Since $R$ is an r-com, $v$ is qualified with respect to $\mathcal{N}_R(v)$, so according to Equation 1 we have $r_v > i = r_u$. This contradicts with our assumption that $r_v$ is the smallest.* □

LEMMA 4. *Given graph $G'$ and schema $S$, vertex $v$ is removed in the $i$-th round in algorithm 1 **if and only if** Equation 1 holds for every vertex and $r_v = i \neq \infty$.*

PROOF. *We prove it by strong induction. First, $\mathcal{N}^1_{G'}(v) = \mathcal{N}_{G'}(v)$, so $r_v = 1$ if and only if $v$ is removed in the first round. Now suppose this holds for $i \leq k$, then the remaining neighbor set of $v$ is exactly $\mathcal{N}^{k+1}_{G'}(v)$. Therefore $r_v = k+1$ if and only if $v$ is removed in the $(k+1)$-th round.* □

COROLLARY 2. *Given $G[Q]$ and $S$, the round number of each vertex is deterministic.*

PROOF. *Algorithm 1 removes vertices synchronously in each round, so within a round, the vertices to remove are fixed. Vertices are also removed at the earliest possible round. Together with Lemma 3 and Lemma 4, the round number of each vertex is deterministic.* □

Given the round index, if $r_q = \infty$, we immediately know that an r-com containing $q$ exists. It is obvious that this round index has space complexity $O(|Q|)$. We next present the initialization and maintenance of the round index, as well as how we apply it in the local search.

### *4.4.1 Index Initialization*

At the beginning of the local search, $Q$ contains a single vertex $q$, so the round index contains a single integer $r_q$. We can simply scan $S$ to determine $r_q$'s value, which is either $\infty$ (if $G[Q]$ is already an r-com) or 1. Therefore, the time cost of initialization is $O(|S|)$.

### *4.4.2 Index Updating*

During the local search approach, vertices are gradually inserted into $Q$, and the correct round number of each vertex would also change. In fact, it can be proved that when inserting/deleting a vertex into/from the graph, no round number will decrease/increase, so we only focus on index increases in this part.

LEMMA 5. *When inserting (resp. deleting) an edge $(w, v)$ in $G[Q]$, no round number will decrease (resp. increase). Specifically, for each vertex $u \in Q$, suppose its round number $r_u$ changes to $r'_u$, then $r_u \leq r'_u$ (resp. $r_u \geq r'_u$).*

PROOF. *We prove the case of inserting an edge by contradiction, and its counterpart can be proved symmetrically. Suppose there is a vertex $u$, such that $r_u > r'_u$.*

*Case 1: $r'_u = 1$, which means $u$ is unqualified even if non of its neighbors is removed. Then because $u$ had a smaller or equal neighbor set before adding $(w, v)$, $r_u$ should also be 1. This contradicts with our assumption.*

*Case 2: $r'_u = t, t > 1$. In this case, since $u$ does not lose any neighbor but $\mathcal{N}^t_{G[Q]}(v)$ changes (Equation 1), there must be a $c \in \mathcal{N}_{G[Q]}(u)$, whose round number $r_c$ decreases to $r'_c$, and $r'_c < t$. If $r'_c > 1$, we can apply the same analysis to $c$ to find its neighbor with a smaller round number. If $r'_c = 1$, it falls into Case 1, which contradicts our assumption.* □

COROLLARY 3. *When inserting (resp. deleting) a vertex $v$ in $Q$, no round number will decrease (resp. increase).*

PROOF. *we prove the case of inserting a vertex, and its counterpart can be proved symmetrically. Inserting a vertex $v$ can be done in 2 steps. First, insert an isolated vertex $v$ into $G[Q]$, and $r_v$ can be evaluated by Equation 1. After this, the round index is still correct for $G[Q]$. Second, insert adjacent edges of $v$ into $G[Q]$ one by one. According to Lemma 5, no round number will decrease.* □

Basically, we handle vertex insertion as follows. When $u$ is inserted into $Q$, we first set $r_u = 0$. Then it could be verified that, among all vertices, Equation 1 does not hold only for $u$, so we use algorithm 6 to evaluate Equation 1 for $u$. Suppose $r_u$ now changes to $x$, $u$'s neighbors may have their round number changed, and so on. To handle all subsequent changes, we devise algorithm 7. Typically, we call "HandleIncrease($u$, 0, $x$)" to handle subsequent changes

**Algorithm 6:** CalcRN

```
  Input  : $G[Q]$, $S$, $v$.
  Output: $r_v$.
1 while $v$ is qualified do
2     $u \leftarrow \text{argmin}_{u \in \mathcal{N}_{G[Q]}(v)} r_u$;
3     $\mathcal{N}_{G[Q]}(v) \leftarrow \mathcal{N}_{G[Q]}(v) \setminus \{u\}$;
4     $r_v \leftarrow r_u + 1$;
5 if $r_v > |Q|$ then $r_v \leftarrow \infty$ ;
6 return $r_v$;
```

induced by $r_u$ changing from 0 to $x$. We next describe the details of algorithm 6 and algorithm 7.

In general, algorithm 6 evaluates Equation 1 by removing $v$'s neighbors round by round. When $v$ becomes unqualified, it will be removed in the next round. A special case is that, when the calculated round is larger than $|Q|$, it returns $\infty$. Its correctness can be proved in Lemma 6, and its time complexity is $O(d_v \cdot \log(d_v))$.

LEMMA 6. *Algorithm 6 correctly evaluates Equation 1 for $v$.*

PROOF. *Lines 1-4 remove $v$'s neighbors round by round, so after line 4 $r_v$ is the smallest round that $v$ becomes unqualified, which is correct. Lines 5-6 deal with a special case when $r_v > |Q|$. When $r_v > |Q|$, we know that there must be a round $k \leq r_v$, in which no vertex is removed, and all remaining vertices belong to r-coms (have round numebr $\infty$). Therefore, vertices have round number larger than $k$ should in fact have round number $\infty$.* □

Algorithm 7 is used to handle all consequent updates induced by the increasing of $r_v$ (which increases from $r_0$ to $r_1$). Overall it runs in a message-passing manner, in which each message $(v, r_0, r_1)$ means "the round number of $v$ is increased from $r_0$ to $r_1$". For each message, it checks every neighbor $u$ of $v$, to see whether $r_u$ may change. According to Equation 1, $r_u$ may increase only if $\mathcal{N}^{r_u}_{G[Q]}(v)$ changes, which means $r_0 < r_u < r_1$. So, in this situation the algorithm calculates the new round number $r'_u$, and then pushes a message if $r_u$ really increases (lines 7-8).

**Algorithm 7:** Handle Increase

```
  Input  : $G[Q]$, $S$, $v$, $r_0$, $r_1$.
1 Queue $\leftarrow \{(v, r_0, r_1)\}$;
2 while $Queue \neq \emptyset$ do
3     ($v'$, $r'_0$, $r'_1$) $\leftarrow$ Queue.pop();
4     foreach $u \in \mathcal{N}_{G[Q]}(v')$ do
5         if $r'_0 < r_u \leq r'_1$ then
6             $r'_u \leftarrow$ CalcRN ($u$);
7             if $r'_u > r_u$ then Queue.push(($u$, $r_u$, $r'_u$)) ;
8             $r_u \leftarrow r'_u$;
```

We prove the correctness of algorithm 7 in Theorem 1. Its time complexity is given in Theorem 2. It should be noted that this complexity is the worst-case complexity. In practice, the updates usually stop quickly because (1) the maximum round number is small, and (2) not all index values are changed.

THEOREM 1. *Algorithm 7 correctly updates the round index induced by a single index value change.*

PROOF. *We have shown that if $r_v$ increases to $r'_v$, there are two possible cases.*

*Case 1: $v$ is a neighbor of the new vertex. This case has been handled when processing the first message.*

*Case 2: $v$'s neighbor $u$ has its round number increased. In this case, there must be a message $\langle u, r_0, r_1 \rangle$ in the queue. The updating of $r_v$ is handled when processing this message.* □

THEOREM 2. *The complexity of algorithm 7 is $O\big(r_{max} \cdot (|E_{G[Q]}| + d_{max} \cdot \log(d_{max})\big)$, where $r_{max}$ is the maximum finite round number in the index, and $d_{max}$ is the maximum degree in $G[Q]$.*

PROOF. *Each vertex can be updated for at most $r_{max}$ times, because each update increases its round number by at least 1. We can use the counting index from Section 3.2 in CalcRN, so that if the round number of a vertex does not changes, line 6 takes only $O(1)$ time. Suppose in the worst case, every vertex is updated for $r_{max}$ times, then the whole algorithm handles $r_{max} \cdot |V_{G[Q]}|$ messages, and reads each edge $r_{max}$ times. The total complexity is thus $O\big(r_{max} \cdot (|E_{G[Q]}| + d_{max} \cdot \log(d_{max})\big)$.* □

Similarly, we can handle index decreases symmetrically with the same time complexity. For simplicity we ignore the details in this paper. We use "HandleIncrease($u$, $r_0$, $r_1$)" and "HandleDecrease($u$, $r_0$, $r_1$)" to denote calls to algorithm 7 and its counterpart, respectively.

### 4.4.3 MRCS with Round Index on Dynamic Graphs

On dynamic graphs, where vertices and edges are frequently inserted and deleted, continuously querying the minimum r-com with the same $q$ and $S$ could be useful. It is expected that previous results can provide some knowledge that helps find the new answer more quickly. In fact, we have shown that the round index supports dynamic updates, so it can naturally handle dynamic graphs.

In this paper, we consider four possible graph changes: vertex insertion/deletion and edge insertion/deletion. Now we discuss how these changes would affect $G[Q]$ and the round index.

- Inserting vertex $v$: Since $v \notin Q$, this operation has no effect on $G[Q]$.
- Deleting vertex $v$: If $v \in Q$, we need to delete $v$ from $G[Q]$; otherwise this operation has no effect on $G[Q]$.
- Inserting (resp. deleting) edge $(u, v)$: If $u, v \in Q$, we need to insert (resp. delete) this edge in $G[Q]$, and update the round index; otherwise this operation has no effect on $G[Q]$.

Following these rules, we use algorithm 8 to handle graph changes. Specifically, we treat deleting a vertex as 2 equivalent operations: deleting all its adjacent edges and then deleting the isolated vertex. In this case, a batch of graph changes can be summarized into two sets $E_a$ and $E_d$, which contain edges to insert and delete, respectively. After handling these changes, the isolated vertices can be removed easily according to discussions in Corollary 3.

In algorithm 8, lines 1-14 modify the graph $G[Q]$ according to $E_a$ and $E_d$, and call "HandleIncrease/HandleDecrease" to deal with index changes. After this, the round index is guaranteed to be correct by Theorem 3. Line 15 deletes vertices that are not reachable from $q$, because they do not belong to any community which contains $q$. Finally, if $r_q \neq \infty$, we need to continue the local search for a new community $R'$ containing $q$.

**Algorithm 8:** Handle Graph Changes

```
   Input : $G[Q]$, $S$, $q$, $E_a$, $E_d$.
1  foreach $(u, v) \in E_a$ do
2      if $u, v \in Q$ then
3          Insert $(u, v)$ into $G[Q]$;
4          if CalcRN $(u) > r_u$ then
5              HandleIncrease ($u$, $r_u$, CalcRN ($u$));
6          if CalcRN $(v) > r_v$ then
7              HandleIncrease ($v$, $r_v$, CalcRN ($v$));
8  foreach $(u, v) \in E_d$ do
9      if $u, v \in Q$ then
10         Delete $(u, v)$ from $G[Q]$;
11         if CalcRN $(u) < r_u$ then
12             HandleDecrease ($u$, $r_u$, CalcRN ($u$));
13         if CalcRN $(v) < r_v$ then
14             HandleDecrease ($v$, $r_v$, CalcRN ($v$));
15 Delete vertices in $G[Q]$ that are not reachable from $q$;
16 Continue Local Search if $r_q \neq \infty$;
```

THEOREM 3. *Algorithm 8 correctly updates the round index given graph changes.*

PROOF. *According to Theorem 1, HandleIncrease and HandleDecrease can correctly update the index after a single index value changes. Therefore, each time an edge $(u, v)$ is inserted or deleted, by calling HandleIncrease or HandleDecrease on u and v (if applicable), the index is correctly updated.* □

Apparently lines 1-14 would call "HandleIncrease" or "HandleDecrease" for at most $2 \cdot (|E_a| + |E_d|)$ times. Line 15 can be done by finding all connected components, which has complexity $O(|E_{G[Q]}|)$. Thus, the overall complexity of algorithm 8 is $O\big(r_{max} \cdot (|E_a| + |E_d|) \cdot (|E_{G[Q]}| + d_{max} \cdot \log(d_{max}))\big)$.

## 5. EXPERIMENTS

In this section we conduct the experimental study on the proposed two problems. We first describe the algorithms, datasets and parameter settings, and then present and analyze experimental results. All algorithms are implemented in C++ and are ran on a machine with an Intel Xeon X5650 CPU and 40GB main memory.

## 5.1 Experimental Setup

### 5.1.1 Datasets

**Graphs.** We test our algorithms on both synthetic and real-world HINs, which are described as follows.

- *IMDB*[1]. This is the MovieLens-100K dataset from [15], which has a simple structure. It contains four types of vertices, i.e., "user" (U), "movie" (M), "actor" (A) and "director" (D). Between vertices are three types of edges, which are "user reviews movie", "actor acts movie" and "director makes movie".
- *Instacart*[2]. This is a co-purchasing network from an online shopping web site. In this network, each vertex is a product that belongs to one of the 21 categories such as "bakery" and "canned". Each edge between two vertices means that they are purchased in the same order for more than 200 times.

[1]https://grouplens.org/datasets/movielens/100k/

[2]https://www.instacart.com/datasets/grocery-shopping-2017

**Table 1: Statistics of Real-World Datasets.**

| | #vertices | #edges | #labels |
|---|---|---|---|
| Instacart | 5,240 | 110,877 | 21 |
| IMDB | 45,913 | 614,580 | 4 |
| YAGO | 3,938,097 | 12,430,701 | 7,124 |
| DIP Hsapi | 3,559 | 13,821 | 59 |

- *YAGO* [10]. YAGO is a large knowledge base derived from many other data sources. Its core graph contains a set of vertices (entities) and edges (facts), where each vertex belongs to one or more categories in the YAGO taxonomy. In our experiments, we label each vertex by the highest-rank category that it belongs to. For example, if a vertex belongs to categories "seafood" and "food", we label it with the higher-rank category "food".
- *DIP Hsapi* [41]. DIP Hsapi is a network of human protein interactions, where each edge represents an interaction between two proteins. The label of each protein can be obtained from the GO database [3]. Each known multi-protein complex, which consists of a set of proteins, is obtained from the MIPS/CORUM database [30] as a ground-truth community. In total there are 468 communities.
- *Synthetic Graphs.* We generate synthetic graphs for scalability test. Specifically, while keeping vertex degrees and labels following two distinct power-law distributions, we vary the number of vertices $|V_G|$ from $10^5$ to $5 \times 10^6$, the average degree $d_G$ from 10 to 40, and the number of labels $|L_G|$ from 10 to 40. By default, we set $|V_G| = 5 \times 10^5$, $d_G = 20$, and $|L_G| = 20$.

The important statistics of real-world datasets are summarized in Table 1.

**Schema.** For each dataset, we randomly generate community schema in the following steps.

1. Randomly pick the concerned set $L_S$;
2. For each pair of labels $(l_1, l_2), l_1, l_2 \in L_S$, generate a random number $x$, and then add constraint $\langle l_1, l_2, x\rangle$ into $S$.

Given a random schema, there may not exist any r-com in the graph, and we only use schema that can query at least one r-com from the graph. Meanwhile, to categorize heterogeneous schema, we define the *cardinality* of a label as the sum of constraints on it, i.e.,

$$Card(l) = \sum_{\langle l, l', c\rangle \in S} c.$$

And then we group schema by their average cardinality, which is denoted by $k$.

**Query Vertices.** In the MRCS problem, for every schema, we randomly select one vertex in an r-com as the query vertex. In this way, we guarantee that each query certainly returns an r-com. Besides, we also conduct experiments to illustrate the performance of each algorithm when the result is empty.

**Graph Updates.** To synthetic dynamic graphs, we randomly insert and delete edges according to the batch size $b$. Specifically, to generate updates of batch size $b$, we randomly select $b/2$ edges in the graph to delete, and $b/2$ pairs of non-neighbor vertices as new edges to insert.

### 5.1.2 Parameter Settings

There are two parameters in our experiments, the average cardinality $k$, and the graph update batch size $b$. For experiments on real-world graphs, we vary $k$ from 2 to 9. For experiments on synthetic graphs, we fix $k = 3$. For experiments on dynamic graphs, we vary $b$ from 10% to 50% of the edge number of the original graph.

### 5.1.3 Algorithms

For the RCD problem, we compare **naive** and message passing (**mp**). For the MRCS problem, we compare four algorithms, which are **exact**, **greedy**, local search (**ls**) and local search with round index (**ls-ri**). For MRCS on dynamic graphs, we report the running time of **ls-ri**.

### 5.1.4 Evaluation Metrics

In efficiency experiments, we evaluate each method by its running time in seconds. For ls-ri, we also record its index size. For each testing, we run each algorithm on 100 different queries, and report the average performance. The maximum running time in all experiments is set to be $10^4$ seconds. Running time exceeding this limit is plotted as "inf".

For effectiveness experiments, we also use two metrics. One is the similarity between two vertices. Given a vertex $v$, we can collect a multi-set of its neighbors' labels $L_{N(v)} = \{\phi(u)|\forall u \in \mathcal{N}_G(v)\}$. Then for two vertices $v, u$ with the same label, we measure their similarity as the jaccard similarity between $L_{N(v)}$ and $L_{N(u)}$. We do not measure the similarity between vertices who have different labels, as it is usually meaningless. In our case study, we borrow the $\text{F}_1$ score from [19] to evaluate the algorithm output against ground-truth communities. Typically, let $\mathcal{C}$ and $\overline{\mathcal{C}}$ be the set of discovered communities and ground-truth communities, respectively, then

$$F_1 = \max_{f:\mathcal{C}\mapsto\overline{\mathcal{C}}} \frac{1}{|f|} \sum_{C\in dom(f)} F_1\big(C, f(C)\big) \tag{2}$$

, where $f$ is a (partial) mapping from $\mathcal{C}$ to $\overline{\mathcal{C}}$.

## 5.2 R-com Effectiveness

### 5.2.1 Node Similarity

We examine the average vertex similarity within r-coms produced by **ls-ri** in this part. As a comparison, we also search the degree-based community [8] (marked as "d-com") with the same query vertex. To make d-coms and r-coms more comparable, we additionally set the cardinality of every label to $k$. In this case, every r-com is also a d-com (i.e., every vertex has degree at least $k$).

We show the results in Figure 4. For the Instacart dataset, we query r-coms among vertices that belong to four categories "bakery" (B), "canned" (C), "house" (H) and "other" (O). For the IMDB dataset, we query r-coms among all types of vertices. We also vary $k$ from 2 to 9 to show the impact of different schema. The missing bars indicate that we cannot calculate the similarity score, which implies that no community contains more than one vertex in that type.

It can be seen that the average vertex similarity in r-coms is consistently higher than that in d-coms. For both methods, when $k$ increases, the score of each vertex type also increase, which shows that setting higher constraints can improve the community quality. The missing bars indicate that there is only one vertex with that type in each community, so no similarity can be calculated.

### 5.2.2 A Case Study on DIP Hsapi

In this part, we examine the quality of r-coms against ground-truth communities in DIP Hsapi. We use the method proposed in Section 2.3 to reverse-engineering 10 schemas from 10 ground-truth communities. Then we further adjust the schema by reducing the largest value in constraints by one, if a schema cannot query more than one community. It turns out that every schema can query more than two communities with at most two adjustments. For the RCD problem, we detect r-coms using these schemas in the network. For the MRCS problem, we run **ls-ri** to query every vertex with each schema and collect all results. In comparison, we also detect $d$-coms in this network by varying all possible $d$. The statistics are shown in Table 2.

**Table 2: Results on DIP Hsapi.**

| | $\|f\|$ per query | | | $F_1$ per query | | |
|---|---|---|---|---|---|---|
| | min | max | mean | min | max | mean |
| d-com | 1 | 3 | 1.6 | 0.02 | 0.72 | 0.20 |
| r-com (RCD) | 2 | 45 | 10.5 | 0.29 | 0.80 | 0.51 |
| r-com (MRCS) | 6 | 157 | 74.2 | 0.41 | 0.67 | 0.51 |

**RCD**. It is shown that on average, r-com beats d-com in not only the number of effective mappings ($|f|$) but also the community quality. Considering that the results of different queries may overlap, we collect all detected r-coms and d-coms to evaluate the overall $F_1$ score. It turns out that all r-coms can be mapped to 92 ground-truth communities with $F_1$ score 0.48, while all d-coms can only be mapped to 8 with a score 0.33. This again shows the effectiveness of r-coms.

**MRCS**. The r-coms of MRCS has competitive quality compared with that of RCD. In fact, we find the most communities using **ls-ri**, and their minimum and mean $F_1$ scores are the highest. All found communities can be mapped to 400 ground-truth communities with an overall $F_1$ score of 0.50, which are both the highest. It turns out that, we can find many more communities with competitive quality by using community search.

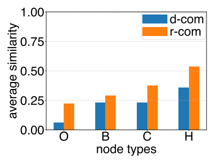
## 5.3 Results on Real-World Graphs

### 5.3.1 Results for RCD

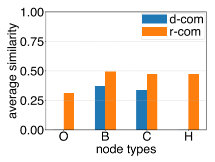
We show the running time of **naive** and **mp** in Figure 5. The maximum construct time and maximum size of the counting index are also reported. Due to its simple structure and low space cost, the counting index takes only a few megabytes of storage and can be constructed in a few seconds. It can be seen that both methods have consistent performance when varying $k$. On smaller datasets, both methods run faster, and **mp** can be an order of magnitude faster than **naive** on the large dataset YAGO.

### 5.3.2 Results for MRCS

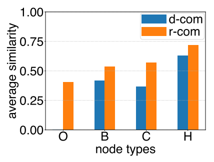
In Figure 6 we show the running time against $k$ and the size of the maximum community containing $q$ ($|R_{max}(q)|$), which is the starting point of **exact** and **greedy**. In addition, we also show the index size ("index") of **ls-ri** to examine its space cost. In general, **exact** fails to finish within the time limit in most cases. The main drawback is its exponential complexity. Other three methods can finish quickly in most cases, while **ls-ri** is the fastest. In fact, **ls-ri** can be orders of magnitude faster than other methods when the search space (index size) is large, which illustrates the efficiency of the round index. Meanwhile, the space cost of the round index is at most hundreds of KBs, which is small compared to the graph size. When looking at (d), (e) and (f), it is easy to notice that the $|R_{max}(q)|$ values on all datasets are large ($> 10^5$). In comparison, the search space of **ls** and



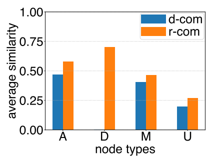
(a) Instacart $k = 2$.



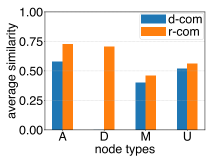
(b) Instacart $k = 5$.



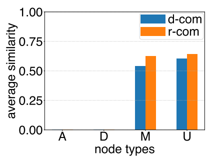
(c) Instacart $k = 9$.



(d) IMDB $k = 2$.

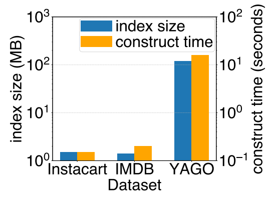


(e) IMDB $k = 5$.

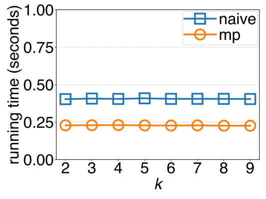


(f) IMDB $k = 9$.

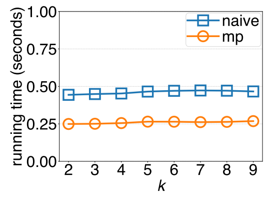
**Figure 4: The average similarity between vertices in each type.**



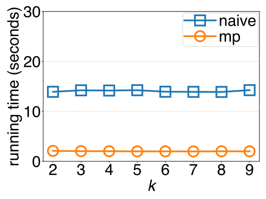
(a) The Counting Index.



(b) Instacart.



(c) IMDB.



(d) YAGO.

**Figure 5: The efficiency for RCD.**

**ls-ri** is $< 10^4$ in most cases. That demonstrates that local search can drastically reduce the search space.

We also show the running time of each method when the returned result is empty in Figure 7. It is clear that the running time of all methods remains to be small in all cases. Particularly, the running time of **ls** and **ls-ri** are close and are significantly smaller than that of **exact** and **greedy**. The reason is that the local search methods usually detect the non-existence of the r-com at an early stage (i.e., when the search space/index size is small) by our candidate pruning strategy, while **exact** and **greedy** have to traverse the whole graph. This shows another advantage of the local search approach of detecting the non-existence of r-coms.

The accuracy of each method in terms of community size is compared in Figure 8. For each method, we record the best community it finds within the time limit, and compare it with the ground-truth ("truth" in the figure). It is clear that **exact** cannot find small communities due to the large search space and time limit, and **greedy** is usually trapped in local optimal. On the other hand, **ls** and **ls-ri** can usually find communities with small size due to the advantage of searching in a small local area. In most cases, their output is close to the ground truth.

We show the update time of **ls-ri** (algorithm 8) on dynamic graphs in Figure 9. It is clear that the update time is short when the graph only changes a bit. As the graph changes more, it gradually approaches the running time on static graphs. Meanwhile, the gap of running time between dynamically updating and running from scratch is larger when the graph is larger, which shows the importance of developing dynamic algorithms for large graphs.

## 5.4 Scalability Test

We test the scalability of all methods on synthetic graphs. The results are shown in Figure 10.

For the RCD problem, both methods can finish within 200 seconds even on large or dense graphs. As $|V_G|$ increases, the running time of two methods increases sub-linearly, because the communities become bigger and the number of removed vertices do not grow quickly. On the other hand, their running time grows linearly to $d_G$, because removing each vertex takes more time on average. When $|L_G|$ is increasing, more vertices should be removed since they do not have the concerned types in $L_S$, so the number of iterations and the running time of **naive** both increase. For **mp**, it is easier to identify a vertex to remove, so the running time decreases a bit.

For the MRCS problem, **exact** and **greedy** do not scale well and fail in almost all cases. When $|V_G|$ increases, the running time of **ls** and **ls-ri** grows sub-linearly, because the search space (index size) does not grow quickly when the graph is large. When $d_G$ is increasing, the search space drops, but the candidate set grows, so the running time of **ls** and **ls-ri** only drops slightly. When $|L_G|$ grows, the connectivity between vertices of the concerned types is sparser, so the search space and running time of **ls** and **ls-ri** both grow. Overall, both **ls** and **ls-ri** scale well when varying $|V_G|$, $d_G$ and $|L_G|$.

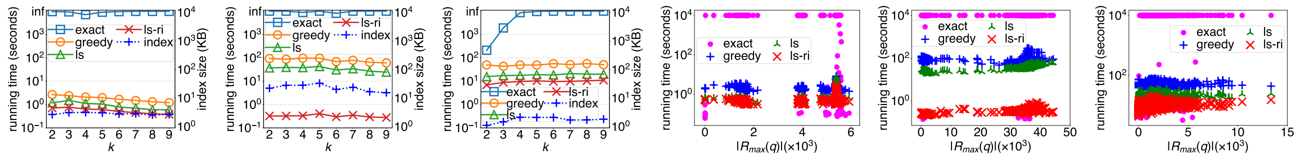
## 6. RELATED WORK

This work is closely related to community detection and community search on graphs.

**Community Detection.** On homogeneous networks, community detection, also known as network clustering, has been extensively studied since it was first introduced in [14]. A variety of methods based on modularity [14], k-clique [24], label-propagation [22,42], correlation [9] are proposed in the literature to improve the quality and efficiency. This kind of method takes only the network structure as input, and use the given model to partition the network into subgraphs.

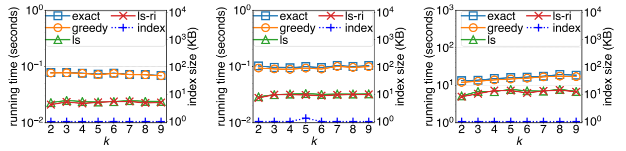
Recently, heterogeneous information network, which integrates semantic information in vertices and edges, has attracted much attention. Zhou et al. [44] propose the SA-Cluster which utilizes the attribute information on edges to improve community quality. Sun et al. [34] propose to integrate incomplete attribute information in clustering. Wang et al. [37] propose a clustering framework that uses text information as indirect supervision, and incorporates subtype information of vertices. User-guided information such as meta-path has been considered in clustering, to provide desired results [36].

Most methods above are designed to detect communities with few user-guided information. They enjoy simple and user-friendly inputs, but fail to provide diverse results at the same time. The only one that takes users' desire into consideration is [36], but how meta-paths control the results is hidden by the probabilistic approach, making it hard for the user to adjust inputs to get better results. In comparison, our proposal in this paper is more suitable when the



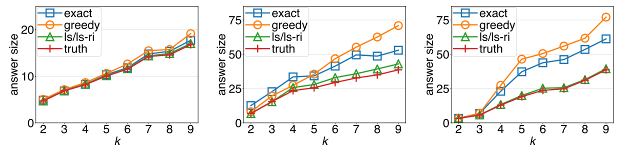
(a) Instacart. (b) IMDB. (c) YAGO. (d) Instacart. (e) IMDB. (f) YAGO.

**Figure 6: The running time for MRCS on static graphs.**



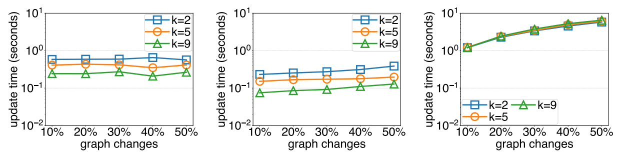
(a) Instacart. (b) IMDB. (c) YAGO.

**Figure 7: The running time for MRCS when the result is empty.**



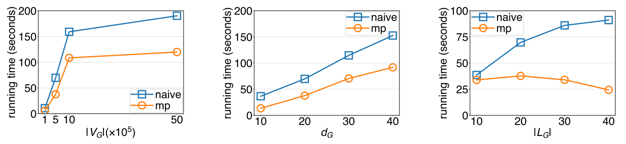
(a) Instacart. (b) IMDB. (c) YAGO.

**Figure 8: The accuracy for MRCS.**

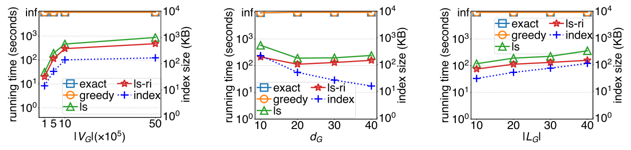


(a) Instacart. (b) IMDB. (c) YAGO.

**Figure 9: The dynamic updating time for MRCS.**



(a) Detection varying $|V_G|$. (b) Detection varying $d_G$. (c) Detection varying $|L_G|$.



(d) Search varying $|V_G|$. (e) Search varying $d_G$. (f) Search varying $|L_G|$.

**Figure 10: Scalability test results.**

user has a clearer picture of what he/she wants, especially for degree requirements.

**Community Search.** One of the main drawbacks of community detection is the need for traversing the whole graph, which has a high cost for large graphs. Meanwhile, the output of community detection might be redundant if the user only cares about a small part of the graph. Considering such inefficiency, methods are developed for searching a (small) community that contains certain vertex(es). Most of them are built upon structure-based models, such as k-core [8,13], k-truss [1,19,21], k-clique [7,40,43] and k-ECC [4,18].

Efforts are spent on integrating heterogeneous information with traditional models. For example, the k-core structure is adapted to incorporate graph attributes like keyword [11], location [12,39], temporal [28], influence [26,27] and profile [5]. There are also methods that adopt k-truss [20] and k-clique [25]. In [11], the user is allowed to provide a set of keywords as part of the input, which implicitly controls the output. Compared to other methods whose input is just one or two parameters, it offers the user more room for specifying and adjusting queries. However, it cannot handle the relational queries proposed in this work.

## 7. CONCLUSION

In this paper, we study the problem of community search in heterogeneous information networks. Specifically, we propose the relational community which is defined upon relational constraints. Using these constraints, the user can specify fine-grained requirements on vertex degrees. We propose efficient algorithms to detect relational communities using message-passing in near optimal-time. For the community search problem, although it is NP-hard, we devise an exact solution and three approximate algorithms, namely greedy, ls and ls-ri. The advantage of ls and ls-ri is that they avoid traversing the whole graph. Moreover, the ls-ri algorithm naturally handles dynamic graphs. We conduct extensive experiments on real-world graphs to show that our proposed methods can provide high-quality results in a short amount of time. Meanwhile, the ls-ri method can handle graph updates efficiently, even with a large number of graph updates.

## 8. ACKNOWLEDGEMENT

Lei Chen's work is partially supported by the Hong Kong RGC GRF Project 16214716, CRF Project C6030-18G, C1031-18G,C5026-18G, AOE Project AoE/E-603/18, China NSFC No. 61729201, Guangdong Basic and Applied Basic Research Foundation 2019B151530001, Hong Kong ITC ITF grants ITS/044/18FX and ITS/470/18FX, Didi-HKUST joint research lab project, Microsoft Research Asia Collaborative Research Grant and Wechat and Webank Research Grants. This work is also supported by the Guangdong Basic and Applied Basic Research Foundation (No. 2019A1515110473).

## 9. REFERENCES

[1] E. Akbas and P. Zhao. Truss-based community search: A truss-equivalence based indexing approach. *PVLDB*, 10(11):1298–1309, 2017.

[2] O. Amini, D. Peleg, S. Pérennes, I. Sau, and S. Saurabh. On the approximability of some degree-constrained subgraph problems. *Discrete Applied Mathematics*, 2012.

[3] E. Camon, M. Magrane, D. Barrell, V. Lee, E. Dimmer, J. Maslen, D. Binns, N. Harte, R. Lopez, and R. Apweiler. The gene ontology annotation (goa) database: sharing knowledge in uniprot with gene ontology. *Nucleic acids research*, 2004.

[4] L. Chang, X. Lin, L. Qin, J. X. Yu, and W. Zhang. Index-based optimal algorithms for computing steiner components with maximum connectivity. In *ACM SIGMOD*, 2015.

[5] Y. Chen, Y. Fang, R. Cheng, Y. Li, X. Chen, and J. Zhang. Exploring communities in large profiled graphs. *TKDE*, 2018.

[6] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu. Efficient core decomposition in massive networks. In *ICDE*, 2011.

[7] W. Cui, Y. Xiao, H. Wang, Y. Lu, and W. Wang. Online search of overlapping communities. In *ACM SIGMOD*, 2013.

[8] W. Cui, Y. Xiao, H. Wang, and W. Wang. Local search of communities in large graphs. In *ACM SIGMOD*, 2014.

[9] L. Duan, W. N. Street, Y. Liu, and H. Lu. Community detection in graphs through correlation. In *SIGKDD*, 2014.

[10] M. Fabian, K. Gjergji, W. Gerhard, et al. Yago: A core of semantic knowledge unifying wordnet and wikipedia. In *WWW*, 2007.

[11] Y. Fang, R. Cheng, Y. Chen, S. Luo, and J. Hu. Effective and efficient attributed community search. *VLDBJ*, 2017.

[12] Y. Fang, Z. Wang, R. Cheng, X. Li, S. Luo, J. Hu, and X. Chen. On spatial-aware community search. *TKDE*, 2018.

[13] Y. Fang, Z. Wang, R. Cheng, H. Wang, and J. Hu. Effective and efficient community search over large directed graphs. *TKDE*, 2018.

[14] M. Girvan and M. E. Newman. Community structure in social and biological networks. *PNAS*, 2002.

[15] F. M. Harper and J. A. Konstan. The movielens datasets: History and context. *ACM TiiS*, 2015.

[16] J. Hopcroft and R. Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. *Commun. ACM*, 1973.

[17] J. Hu, R. Cheng, K. C.-C. Chang, A. Sankar, Y. Fang, and B. Y. Lam. Discovering maximal motif cliques in large heterogeneous information networks. In *ICDE*, 2019.

[18] J. Hu, X. Wu, R. Cheng, S. Luo, and Y. Fang. On minimal steiner maximum-connected subgraph queries. *TKDE*, 2017.

[19] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu. Querying k-truss community in large and dynamic graphs. In *ACM SIGMOD*, 2014.

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

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

[22] X. Jian, X. Lian, and L. Chen. On efficiently detecting overlapping communities over distributed dynamic graphs. In *ICDE*, 2018.

[23] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*. 1972.

[24] J. M. Kumpula, M. Kivelä, K. Kaski, and J. Saramäki. Sequential algorithm for fast clique percolation. *Physical Review E*, 2008.

[25] J. Li, X. Wang, K. Deng, X. Yang, T. Sellis, and J. X. Yu. Most influential community search over large social networks. In *ICDE*, 2017.

[26] 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 *ACM SIGMOD*, 2018.

[27] R.-H. Li, L. Qin, J. X. Yu, and R. Mao. Finding influential communities in massive networks. *VLDBJ*, 2017.

[28] R.-H. Li, J. Su, L. Qin, J. X. Yu, and Q. Dai. Persistent community search in temporal networks. In *ICDE*, 2018.

[29] L. Qin, R.-H. Li, L. Chang, and C. Zhang. Locally densest subgraph discovery. In *SIGKDD*, 2015.

[30] A. Ruepp, B. Brauner, I. Dunger-Kaltenbach, G. Frishman, C. Montrone, M. Stransky, B. Waegele, T. Schmidt, O. N. Doudieu, V. Stümpflen, et al. Corum: the comprehensive resource of mammalian protein complexes. *Nucleic acids research*, 2007.

[31] J. Shang, J. Shen, L. Liu, and J. Han. Constructing and mining heterogeneous information networks from massive text. In *SIGKDD*, 2019.

[32] C. Shi, X. Kong, P. S. Yu, S. Xie, and B. Wu. Relevance search in heterogeneous networks. In *EDBT*, 2012.

[33] A. Spitz, D. Costa, K. Chen, J. Greulich, J. Geiß, S. Wiesberg, and M. Gertz. Heterogeneous subgraph features for information networks. In *ACM GRADES*, 2018.

[34] Y. Sun, C. C. Aggarwal, and J. Han. Relation strength-aware clustering of heterogeneous information networks with incomplete attributes. *PVLDB*, 5(5):394–405, 2012.

[35] Y. Sun, J. Han, X. Yan, P. S. Yu, and T. Wu. Pathsim: Meta path-based top-k similarity search in heterogeneous information networks. *PVLDB*, 4(11):992–1003, 2011.

[36] Y. Sun, B. Norick, J. Han, X. Yan, P. S. Yu, and X. Yu. Pathselclus: Integrating meta-path selection with user-guided object clustering in heterogeneous information networks. *TKDD*, 2013.

[37] C. Wang, Y. Song, A. El-Kishky, D. Roth, M. Zhang, and J. Han. Incorporating world knowledge to document clustering via heterogeneous information networks. In *SIGKDD*, 2015.

[38] J. Wang and J. Cheng. Truss decomposition in massive networks. *PVLDB*, 5(9):812–823, 2012.

[39] K. Wang, X. Cao, X. Lin, W. Zhang, and L. Qin. Efficient computing of radius-bounded k-cores. In *ICDE*, 2018.

[40] Y. Wang, X. Jian, Z. Yang, and J. Li. Query optimal k-plex based community in graphs. *DSE*, 2017.

[41] I. Xenarios, L. Salwinski, X. J. Duan, P. Higney, S.-M. Kim, and D. Eisenberg. Dip, the database of interacting proteins: a research tool for studying cellular networks of protein interactions. *Nucleic acids research*, 2002.

[42] J. Xie and B. K. Szymanski. Towards linear time overlapping community detection in social networks. In *PAKDD*, 2012.

[43] L. Yuan, L. Qin, W. Zhang, L. Chang, and J. Yang. Index-based densest clique percolation community search in networks. *TKDE*, 2017.

[44] Y. Zhou, H. Cheng, and J. X. Yu. Graph clustering based on structural/attribute similarities. *PVLDB*, 2(1):718–729, 2009.