S^{3}AND: Efficient Subgraph Similarity Search Under Aggregated Neighbor Difference Semantics

Qi Wen
East China Normal University
Shanghai, China
51265902057@stu.ecnu.edu.cn

Yutong Ye
East China Normal University, Shanghai, China
Kent State University, Kent, Ohio, USA
52205902007@stu.ecnu.edu.cn

Xiang Lian
Kent State University
Kent, Ohio, USA
xlian@kent.edu

Mingsong Chen
East China Normal University
Shanghai, China
mschen@sei.ecnu.edu.cn

ABSTRACT

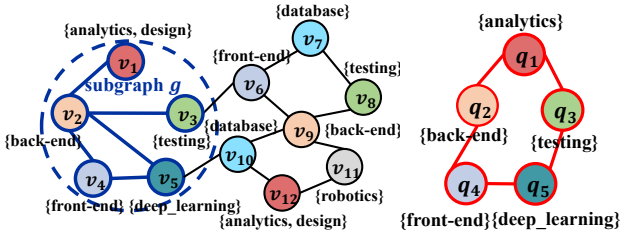
For the past decades, the *subgraph similarity search* over a large-scale data graph has become increasingly important and crucial in many real-world applications, such as social network analysis, bioinformatics network analytics, knowledge graph discovery, and many others. While previous works on subgraph similarity search used various graph similarity metrics such as the graph isomorphism, graph edit distance, and so on, in this paper, we propose a novel problem, namely *subgraph similarity search under aggregated neighbor difference semantics* (S^{3}AND), which identifies subgraphs g in a data graph G that are similar to a given query graph q by considering both keywords and graph structures (under new keyword/structural matching semantics). To efficiently tackle the S^{3}AND problem, we design two effective pruning methods, *keyword set* and *aggregated neighbor difference lower bound pruning*, which rule out false alarms of candidate vertices/subgraphs to reduce the S^{3}AND search space. Furthermore, we construct an effective indexing mechanism to facilitate our proposed efficient S^{3}AND query answering algorithm. Through extensive experiments, we demonstrate the effectiveness and efficiency of our S^{3}AND approach over both real and synthetic graphs under various parameter settings.

PVLDB Reference Format:
Qi Wen, Yutong Ye, Xiang Lian, and Mingsong Chen. S^{3}AND: Efficient Subgraph Similarity Search Under Aggregated Neighbor Difference Semantics. PVLDB, 18(11): 3708-3720, 2025.
doi:10.14778/3749646.3749648

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Luminous-wq/S3AND.

1 INTRODUCTION

The *subgraph similarity search* over graphs has been widely used as an important and fundamental tool for real-world applications, such as social network analysis [28], knowledge graph discovery [29], bioinformatics mining [9], and so on. Specifically, a subgraph similarity search query retrieves those subgraphs g in a large-scale data graph G that are similar to a given query graph pattern q.



(a) collaboration social network G (b) target team (query graph) q

Figure 1: An S^{3}AND example of the skilled team search.

Existing works on the subgraph similarity search used graph similarity metrics, such as *graph edit distance* [5, 8] and *chi-square statistics* [7], to measure the similarity between subgraphs g and query graph q. While different graph similarity semantics are helpful for different real applications (e.g., with similar graph structures or statistics), in this paper, we propose a novel graph similarity measure, called *aggregated neighbor difference* (AND), which is given by aggregating the neighbor differences of the matching vertices between subgraph g and query graph q. Based on this AND semantic, we formulate a new problem, namely *subgraph similarity search under aggregated neighbor difference semantics* (S^{3}AND), which obtains subgraphs $g \subseteq G$ that match with q with low AND scores.

Below, we give a motivation example of our S^{3}AND problem in the application of collaboration social network analysis.

Example 1. ***(The Skilled Team Search in Collaboration Social Network)*** *To accomplish a new project, a manager wants to recruit an experienced team that consists of members with relevant skills and previous collaboration experiences. Figure 1(a) shows a collaboration social network G, which consists of 12 user vertices, $v_1 \sim v_{12}$, each with a set of skill labels (e.g., user v_2 has the "back-end" development skills), and collaborative edges (each connecting two users, e.g., v_2 and v_3, indicating that they collaborated on some project before).*

Figure 1(b) shows a target (query) graph pattern q, which represents a desirable team structure, specified by the project manager. In particular, each member q_i ($1 \le i \le 5$) in this experienced team must have certain skills (i.e., query keywords), for example, team member q_2 should have the "back-end" skill. Moreover, during the project period, team members are required to communicate frequently for accomplishing the project together. Thus, it is preferred that they have

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749648

previous experience in project collaboration to reduce the one-to-one communication overhead. As an example, the "back-end" team member q_2 is expected to have collaborative experience (i.e., edge $e(q_2, q_4)$) with a "front-end" member q_4 before.

However, in practice, it is rare to find a perfect subgraph that exactly matches the query graph q. For example, in Figure 1, we cannot find a subgraph of G that is structurally isomorphic to the query graph q.

Alternatively, the manager can issue an S^{3}AND query to obtain a team from G (e.g., the subgraph g within the dashed circle of Figure 1(a)), whose members have the required skills to accomplish project tasks (i.e., data vertices in g must contain the required keywords in query vertices), but follow some relaxed constraints on the collaboration experience. As an example, in subgraph g of Figure 1(a), although user v_1 does not have experience working with user v_3 before (as required by edge $e(q_1, q_3)$ in query graph q), they can still build good collaborative relationships through the new project, however, with some extra costs (e.g., time delays and/or communication efforts, as they were not familiar with each other before). Similarly, compared with the target team q, collaborative edge $e(v_3, v_5)$ is also missing in g. Thus, although subgraph g and query graph q do not structurally match with each other, subgraph g can still be a potential candidate team that follows strict skill constraints and meets the relaxed collaboration requirements (e.g., within some budget of extra collaboration costs, defined later as the aggregated neighbor differences (AND) in Section 2.2). The S^{3}AND query can exactly help obtain such a team (subgraph) g in G, satisfying the keyword set containment relationship between query and data vertices and with low collaboration/communication overheads (i.e., AND scores). ■

The S^{3}AND problem has many other real applications. For example, in the Semantic Web applications [44], a SPARQL query can be considered as a query graph q over a large knowledge graph G. Our S^{3}AND query can be used to return RDF subgraphs that follow the keyword constraints and have minor structural changes compared with q.

Inspired by the examples above, our S^{3}AND problem considers novel *aggregated neighbor difference* (AND) semantics for subgraph similarity search over a large data graph G. Efficient and effective S^{3}AND query answering is quite challenging, due to complex graph manipulations (e.g., graph structure/keyword checking and AND score calculation over a large-scale data graph). Therefore, in this paper, we will design a general framework for S^{3}AND query processing, which seamlessly integrates our proposed effective pruning strategies (with respect to keywords and AND scores) to reduce the problem search space, effective indexing mechanisms over precomputed data from graph G, and efficient S^{3}AND query algorithm via the index traversal.

Specifically, we make the following contributions in this paper.

(1) We formulate a novel problem, *subgraph similarity search under aggregated neighbor difference semantics* (S^{3}AND) in Section 2, which is useful for real application scenarios.
(2) We propose a general framework for tackling our S^{3}AND problem efficiently and effectively in Section 3.
(3) We design effective pruning strategies (w.r.t. constraints of keywords and aggregated neighbor differences) in Section 4 to prune false alarms of candidate vertices/subgraphs.
(4) We devise an effective indexing mechanism to facilitate our proposed S^{3}AND processing algorithm in Section 6.
(5) We validate the S^{3}AND performance in Section 7 through extensive experiments on real/synthetic graphs.

Section 8 overviews previous works on subgraph matching and subgraph similarity search. Finally, Section 9 concludes this paper.

2 PROBLEM DEFINITION

2.1 Graph Data Model

We first provide the formal definition of a large-scale data graph G.

Definition 1. ***(Data Graph, G)*** *A data graph G is in the form of a triple $(V(G), E(G), \Phi(G))$, where $V(G)$ is a set of vertices, v_i, in graph G, each with a keyword set $v_i.W$, $E(G)$ represents a set of edges $e(v_i, v_j)$ (connecting two ending vertices v_i and v_j), and $\Phi(G)$ is a mapping function: $V(G) \times V(G) \rightarrow E(G)$.*

2.2 Aggregated Neighbor Difference Semantics

The Vertex-to-Vertex Mapping, $M : V(q) \rightarrow V(g)$: Consider a target (query) graph pattern q and a subgraph g of data graph G with the same graph size, that is, $|V(g)| = |V(q)|$. We say that there is a vertex-to-vertex mapping, $M : V(q) \rightarrow V(g)$, between q and g, if each vertex $q_j \in V(g)$ has a 1-to-1 mapping to a query vertex $q_j \in V(q)$, such that their keyword sets satisfy the condition that $q_j.W \subseteq v_i.W$.

The Vertex Subset Mapping Function, $\mu(\cdot)$: Accordingly, we denote $\mu(\cdot)$ as a mapping function from any vertex subset, $V'(q)$, of $V(q)$ to its mapping subset, $V'(g)$, of $V(g)$ (via the vertex-to-vertex mapping M). That is, we have $\mu(V'(q)) = V'(g)$, where any vertex $q_j \in V'(q)$ is mapped to a vertex $M(q_j) = v_i \in V'(g)$.

Neighbor Difference Semantics, $ND(q_j, v_i)$: Let $N(v_i)$ be a set of 1-hop neighbors of vertex $v_i \in V(g)$ in the subgraph g. Similarly, $N(q_j)$ is a set of q_j's 1-hop neighbors in the query graph q.

Then, for each vertex pair (v_i, q_j) between g and q, their *neighbor difference*, $ND(q_j, v_i)$, is defined as the number of (matching) 1-hop neighbors (or edges) that v_i is missing, based on the target vertex q_j (and its neighbors). Formally, we have the following definition of the neighbor difference semantics.

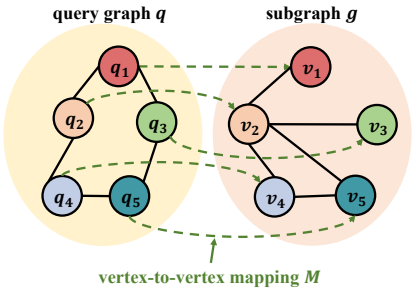
Definition 2. ***(Neighbor Difference, $ND(q_j, v_i)$)*** *Given a target vertex, q_j, of a query graph q, a vertex, v_i, of a subgraph g, and a mapping function $\mu(\cdot)$ from any subset of $V(q)$ to its corresponding subset of $V(g)$ (via vertex-to-vertex mapping M), their neighbor difference, $ND(q_j, v_i)$, is given by:*

$$ND(q_j, v_i) = |\mu(N(q_j)) - N(v_i)|, \quad (1)$$

where $N(\cdot)$ is a set of 1-hop neighbor vertices, "$-$" is a set difference operator, and $|\cdot|$ is the cardinality of a set.

Intuitively, in Definition 2, for each vertex pair (q_j, v_i) in q and g, the neighbor difference, $ND(q_j, v_i)$, is given by the number of missing edges $e(v_i, v_j)$ with an ending vertex v_i (when their corresponding edges $e(q_j, q_j)$ in the target query graph q exist).

Example 2. *(Continue with Example 1). In the previous example of Figure 1, compared with the query vertex q_1 in query graph q, the data vertex v_1 in subgraph g has one missing edge between v_1 and v_3 (while edge $e(q_1, q_3)$ exists in q). Thus, we have the neighbor difference $ND(q_1, v_1) = 1$. Similarly, since vertex v_2 has 1-hop neighbors v_1 and v_4 (while edges $e(q_2, q_1)$ and $e(q_2, q_4)$ exist in q), we have $ND(q_2, v_2) = 0$.* ■



(a) query graph q and its matching subgraph g

query vertex $q_i \in V(q)$	missing edges $e(v_i, v_j)$ in subgraph g	neighbor difference $ND(q_i, v_i)$	aggregated neighbor difference $AND(q, g)$
q_1	$e(v_1, v_3)$	$ND(q_1, v_1) = 1$	$MAX: 2$
q_2	–	$ND(q_2, v_2) = 0$	
q_3	$e(v_3, v_1), e(v_3, v_5)$	$ND(q_3, v_3) = 2$	$AVG: 0.8$
q_4	–	$ND(q_4, v_4) = 0$	
q_5	$e(v_5, v_3)$	$ND(q_5, v_5) = 1$	$SUM: 4$

(b) aggregated neighbor difference between q and g

Figure 2: An Example of the Aggregated Neighbor Difference.

The Aggregation Over Neighbor Differences, $AND(q, g)$: Next, we consider the *aggregated neighbor differences* (AND), $AND(q, g)$, for vertex pairs (q_j, v_i) from query graph q and subgraph g, respectively. Intuitively, $AND(q, g)$ outputs an aggregation over the numbers of missing edges $e(v_i, v_j)$ (or 1-hop neighbors v_j in $N(v_i)$) for all vertices v_i in g, according to the targeted query graph q (i.e., edges $e(q_j, q_j)$ in q).

Definition 3. ***(Aggregated Neighbor Difference, $AND(q, g)$)*** *Given a query graph q, a subgraph g, and a 1-to-1 vertex mapping M from $V(q)$ to $V(g)$ (note: $|V(q)| = |V(g)|$), the aggregated neighbor difference, $AND(q, g)$, between q and g is defined as the aggregation over neighbor differences of all the matching vertex pairs (q_j, v_i), i.e.,*

$$AND(q, g) = f\left(\left\{ND(q_j, v_i) | \forall (q_j, v_i), s.t. M(q_j) = v_i\right\}\right), \quad (2)$$

where $ND(q_j, v_i)$ is given by Eq. (1), and $f(S)$ is an aggregate function (e.g., MAX, AVG, or SUM) over a set S.

In Definition 3, the aggregated neighbor difference, $AND(q, g)$, is given by the aggregation over neighbor differences $ND(q_j, v_i)$ of all the matching pairs (q_j, v_i) between q and v_i. The aggregation function $f(S)$ can have different semantics such as MAX, AVG, or SUM. In Example 1 (i.e., the skilled team search), the MAX aggregate function returns the *maximum* possible collaboration effort (i.e., $ND(q_j, v_i)$) that team members v_i need (due to no collaboration experience with other team members before). Similarly, AVG (or SUM) aggregate function obtains the extra collaboration cost each team member has to spend on average (or the total collaboration cost for the entire team).

Example 3. *Figure 2(a) illustrates the vertex-to-vertex mapping M from vertices of query graph q to that of subgraph g (as given in Example 1), whereas Figure 2(b) shows the neighbor differences, $ND(q_j, v_i)$, in subgraph g, for each query vertex q_j, and their aggregated neighbor differences $AND(q, g)$ under different semantics.*

Specifically, in Figure 2(b), we can see that vertex v_1 is not connected to v_3 in g, compared with the edge $e(q_1, q_3)$ in the target query graph q. Thus, we have $ND(q_1, v_1) = 1$. Similarly, we can compute neighbor differences for other vertex pairs (q_j, v_i) (for $i \geq 2$). By aggregating these neighbor differences $ND(q_j, v_i)$ (for $1 \leq i \leq 5$) with different aggregation functions $f(\cdot)$ (as given in Definition 3), we can obtain their aggregated neighbor difference, $AND(q, g)$, that is, 2 ($= \max\{1, 0, 2, 0, 1\}$) for MAX, 0.8 ($= \frac{1+0+2+0+1}{5}$) for AVG, and 4 ($= 1 + 0 + 2 + 0 + 1$) for SUM. ■

Note that, since the AND score $AND(q, g)$ with the AVG aggregate is given by the AND score with the SUM aggregate divided by a constant (i.e., $|V(q)|$). In subsequent discussions, we will only focus on MAX and SUM aggregates for f (note: AVG has the same S^{3}AND query answers as SUM).

2.3 The S^{3}AND Problem Definition

In this subsection, we formulate the *subgraph similarity search problem under the aggregated neighbor difference semantics* (S^{3}AND).

Definition 4. ***(Subgraph Similarity Search Under Aggregated Neighbor Difference Semantics, S^{3}AND(G, q))*** *Given a data graph G, a query graph q, a vertex-to-vertex mapping $M : V(q) \rightarrow V(g)$, and an aggregation threshold σ, a subgraph similarity search under the aggregated neighbor difference semantics (S^{3}AND) retrieves connected subgraphs g of G, such that:*

- **(Equal Subgraph Size)** $|V(q)| = |V(g)|$;
- **(Keyword Set Containment)** *for the mapping vertices $q_j \in V(q)$ and $v_i \in V(g)$, it holds that $q_j.W \subseteq v_i.W$, and;*
- **(Aggregated Neighbor Difference)** *the aggregated neighbor difference satisfies the condition that $AND(q, g) \leq \sigma$,*

where $AND(q, g)$ is given by Eq. (2).

Intuitively, in Definition 4, the S^{3}AND problem retrieves all the subgraphs g that satisfy the AND constraints, with respect to the query graph q. In particular, there exists a 1-to-1 vertex mapping, M, from each subgraph g to query graph q. Thus, they have equal graph size, that is, $|V(q)| = |V(g)|$. Moreover, for the mapping vertices q_j and v_i from graphs q and g, respectively, their associated keyword sets satisfy the containment relationship, that is, $q_j.W \subseteq v_i.W$. Further, their aggregated neighbor difference $AND(q, g)$ should be as low as possible (i.e., $AND(q, g) \leq \sigma$). The three conditions above guarantee that the subgraphs g can maximally match with the required target graph pattern q.

In Definition 4, we used the constraint of the *Keyword Set Containment*. In practice, we may also consider other keyword matching constraints such as keyword embedding similarity, ontology similarity, and so on, and adapt our proposed techniques (e.g., pruning and indexing) to handle such keyword matching constraints. Moreover, the AND score, $AND(q, g)$, considers the (mis)matching of edges between query/data vertices and their 1-hop neighbors. As in Example 1, the AND score implies the communication overhead between team members and their collaborators (i.e., 1-hop neighbors in collaboration networks). We would like to leave interesting topics of considering variants of S^{3}AND query semantics (e.g., with different keyword matching or topological similarity options) as our future work.

Table 1 depicts the commonly used notations and their descriptions in this paper.

3 THE S^{3}AND PROCESSING FRAMEWORK

Algorithm 1 illustrates a general framework for S^{3}AND query answering in a large-scale data graph G. Figure 3 provides a visual

Table 1: Symbols and Descriptions

Symbol	Description
G	a data graph
$V(G)$	a set of vertices v_i
$E(G)$	a set of edges $e(u, v)$
q	a query graph
g	a subgraph of data graph G
$v_i.W$	a keyword set of vertex v_i
$N(v_i)$	a set of vertex v_i's 1-hop neighbors
$ND(q_j, v_i)$	the 1-hop neighbor difference between vertices q_j and v_i
$AND(q, g)$	the aggregation over all the neighbor differences of vertex pairs (q_j, v_i) in graphs q and g
σ	a threshold for the aggregated neighbor difference

workflow of the pseudo code in Algorithm 1, which consists of two phases, *offline pre-computation* (lines 1-3 of Algorithm 1) and *online S^3AND query processing phases* (lines 4-7 of Algorithm 1).

Specifically, as illustrated in Figure 3, in the *offline pre-computation phase*, we offline pre-compute some auxiliary data, $v_i.Aux$, of each vertex v_i in large-scale data graph G (lines 1-2 of Algorithm 1), and construct a tree index $\mathcal{I}$ over these pre-computed data $v_i.Aux$ to facilitate online query optimizations like pruning (line 3 of Algorithm 1).

In the *online S^3AND query processing phase*, for each S^3AND query, we traverse the tree index $\mathcal{I}$ by applying our proposed pruning strategies (e.g., the keyword set and AND lower bound pruning) to retrieve candidate vertices w.r.t. query vertices q_j in the query graph q (lines 4-5 of Algorithm 1). Next, we assemble candidate vertices of query vertices q_j and obtain candidate subgraphs g (line 6 of Algorithm 1). Finally, we refine candidate subgraphs g and return a set, S, of actual S^3AND subgraph answers (line 7 of Algorithm 1).

4 PRUNING STRATEGIES

In this section, we present effective pruning strategies that reduce the problem search space during the online S^3AND query processing phase (lines 5-7 of Algorithm 1). **Due to space limitations, we report proofs of lemmas throughout this paper in our technical report [37] and omit them below.**

4.1 Keyword Set Pruning

In Definition 4, the keyword set $v_i.W$ of each vertex v_i in the S^3AND subgraph answer g must be a superset of the keyword set $q_j.W$ for its corresponding query vertex q_j in the query graph q. Based on this, we design an effective *keyword set pruning* method to rule out candidate vertices that do not satisfy this keyword set constraint.

Lemma 1. **(Keyword Set Pruning)** *Given a candidate vertex v_i and a query graph q, vertex v_i can be safely pruned, if it holds that: $v_i.W \cap q_j.W \neq q_j.W$ (i.e., $q_j.W \not\subseteq v_i.W$), for all $q_j \in V(q)$.*

Discussions on How to Implement the Keyword Set Pruning: In order to enable the keyword set pruning (as given in Lemma 1), we can offline pre-compute a bit vector, $v_i.BV$, of size B for the keyword set $v_i.W$ in each vertex v_i. In particular, we first initialize the bit vector $v_i.BV$ with $\vec{0}$, and then hash each keyword in $v_i.W$ into a bit position in $v_i.BV$ (via a hashing function; setting the position to "1"). The case of computing query bit vector $q_j.BV$ for query keyword set $q_j.W$ (w.r.t. query vertex q_j) is similar.

As a result, the pruning condition, $v_i.W \cap q_j.W \neq q_j.W$, in the keyword set pruning method can be written as:

$$v_i.BV \ \wedge \ q_j.BV \neq q_j.BV, \tag{3}$$

where "$\wedge$" is a bit-AND operator between two bit vectors.

Enhancing the Pruning Power via Keyword Grouping: Since the keyword domain of the real data may be large, the size, B, of bit vectors $v_i.BV$ is much smaller than the keyword domain size, which may lead to hashing conflicts (i.e., different keywords are hashed to the same bit position in $v_i.BV$). In order to enhance the pruning power of keyword set pruning, we propose a *keyword grouping* optimization approach, which can reduce the probability of incurring false positives via keyword set pruning.

Specifically, we divide the keyword domain into m disjoint groups. For each vertex v_i, if a keyword in $v_i.W$ falls into the x-th keyword group, we will hash this keyword into the x-th bit vector $v_i.BV^{(x)}$ (for $1 \leq x \leq m$) via a hashing function. This way, the pruning condition, $v_i.W \cap q_j.W \neq q_j.W$, in the keyword set pruning (Lemma 1) can be rewritten as:

$$\bigvee_{x=1}^{m} \left(v_i.BV^{(x)} \ \bigwedge \ q_j.BV^{(x)} \neq q_j.BV^{(x)} \right), \tag{4}$$

where $v_i.BV^{(x)}$ and $q_j.BV^{(x)}$ are bit vectors with the hashed keywords from the x-th keyword group in $v_i.W$ and $q_j.W$, respectively.

Algorithm 1: The S^3AND Processing Framework

Input: i) a data graph G, ii) a query graph q, and iii) an aggregated neighbor difference threshold σ
Output: a set, S, of subgraphs g matching with the query graph q under AND semantics

// **offline pre-computation phase**
1 **for** *each* $v_i \in V(G)$ **do**
2 compute the auxiliary data $v_i.Aux$
3 construct a tree index $\mathcal{I}$ over pre-computed aggregate data in graph G
// **online S^3AND query processing phase**
4 **for** *each S^3AND query* **do**
5 traverse the tree index $\mathcal{I}$ by applying the keyword set and AND lower bound pruning strategies to retrieve candidate vertices w.r.t. query vertices q_j in the query graph q
6 assemble candidate vertices of query vertices q_j and obtain candidate subgraphs g
7 refine candidate subgraphs g and return a set, S, of actual S^3AND subgraph answers

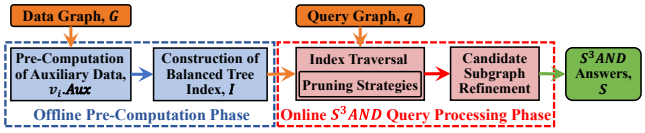


Figure 3: The workflow of S^3AND query processing.

4.2 AND Lower Bound Pruning

According to Definition 4, the aggregated neighbor difference (AND) between a subgraph g and a query graph q must satisfy the AND constraint, that is, $AND(q, g) \leq \sigma$. Therefore, we present an *AND lower bound pruning* method, which effectively filters out candidate subgraphs with high AND values below.

Lemma 2. **(AND Lower Bound Pruning)** *Given a candidate subgraph g, a query graph q, and an aggregated neighbor difference threshold σ, subgraph g can be safely pruned, if it holds that $lb\_AND(q, g) > \sigma$, where $lb\_AND(q, g)$ is a lower bound of $AND(q, g)$.*

Discussions on How to Compute an AND Lower Bound, $lb\_AND(q,g)$: Based on Eq. (2), in order to compute a lower bound, $lb\_AND(q,g)$, of the AND score $AND(q,g)$, we only need to obtain a lower bound, $lb\_ND(q_j,v_i)$, of the neighbor difference $ND(q_j,v_i)$ (as given in Eq. (1)) for each matching vertex pair (q_j,v_i). This way, we have:

$$lb\_AND(q,g) = f\left(\left\{lb\_ND(q_j,v_i) | \forall (q_j,v_i), s.t. M(q_j)=v_i\right\}\right). \quad (5)$$

The Computation of the Neighbor Difference Lower Bound $lb\_ND(q_j,v_i)$. To compute a lower bound $lb\_ND(q_j,v_i)$ of the neighbor difference $ND(q_j,v_i)$, we can rewrite the neighbor difference $ND(q_j,v_i)$ in Eq. (1) as follows:

$$\begin{aligned} ND(q_j,v_i) &= |\mu(N(q_j)) - N(v_i)| \\ &= |\mu(N(q_j))| - |\mu(N(q_j)) \cap N(v_i)|. \end{aligned} \quad (6)$$

In Eq. (6), the first term $|\mu(N(q_j))|$ is a constant during online S^3AND query processing (i.e., the number of vertex q_j's neighbors in the query graph q). Thus, in order to calculate the ND lower bound $lb\_ND(q_j,v_i)$, we alternatively need to compute an upper bound of the second term in Eq. (6) (i.e., $|\mu(N(q_j)) \cap N(v_i)|$). Since $N(v_i) \supseteq (\mu(N(q_j)) \cap N(v_i))$ holds, we can obtain its upper bound: $|N(v_i)| \geq |\mu(N(q_j)) \cap N(v_i)|$.

In other words, we have the ND lower bound below:

$$lb\_ND(q_j,v_i) = \max\{0, |\mu(N(q_j))| - |N(v_i)|\}. \quad (7)$$

The Computation of a Tighter Neighbor Difference Lower Bound $lb\_ND(q_j,v_i)$. Note that, some neighbors, q_l, of query vertex q_j may not match with that, v_l, of data vertex v_i with respect to their keyword sets (i.e., $q_l.W \not\subseteq v_l.W$). Therefore, $|N(v_i)|$ may not be a tight upper bound of $|\mu(N(q_j)) \cap N(v_i)|$, and in turn $lb\_ND(q_j,v_i)$ in Eq. (7) is not a tight neighbor difference lower bound.

Below, we will consider the keyword set matching between (neighbors of) vertices v_i and q_j, and derive a tighter neighbor difference lower bound. Specifically, for each neighbor q_l of query vertex q_j, if its keyword set $q_l.W$ is a subset of the union of keyword sets from $N(v_i)$ (i.e., v_i's neighbors), we can count 1, for the upper bound of $|\mu(N(q_j)) \cap N(v_i)|$). Formally, we have this upper bound:

$$\sum_{q_l \in N(q_j)} \Phi\left(q_l.W \subseteq \cup_{\forall v_l \in N(v_i)} v_l.W\right) \geq |\mu(N(q_j)) \cap N(v_i)|,$$

where $\Phi(\cdot)$ is an indicator function (i.e., $\Phi(z) = 1$, if z is *true*; $\Phi(z) = 0$, otherwise).

As a result, we can obtain a tighter ND lower bound below:

$$lb\_ND(q_j,v_i) = |\mu(N(q_j))| - \sum_{q_l \in N(q_j)} \Phi\left(q_l.W \subseteq \cup_{\forall v_l \in N(v_i)} v_l.W\right). \quad (8)$$

To efficiently check the containment of two keyword sets in Eq. (8) (i.e., $q_l.W \subseteq \cup_{\forall v_l \in N(v_i)} v_l.W$), we can also use their keyword bit vectors to replace the parameter of the indicator function $\Phi(\cdot)$ in Eq. (8), that is,

$$\begin{aligned} lb\_ND(q_j,v_i) &= |\mu(N(q_j))| \\ &- \sum_{q_l \in N(q_j)} \Phi\left(\bigwedge_{x=1}^{m}\left(q_l.BV^{(x)} \bigwedge\left(\bigvee_{\forall v_l \in N(v_i)} v_l.BV^{(x)}\right) = q_l.BV^{(x)}\right)\right), \end{aligned} \quad (9)$$

where $\wedge$ and $\vee$ are bit-AND and bit-OR operators between two bit vectors, respectively.

Algorithm 2: Offline Pre-Computation of Auxiliary Data

Input: i) a data graph G, and ii) the number, m, of keyword groups
Output: the pre-computed auxiliary data $v_i.Aux$ for each vertex v_i

1 **for** *each vertex* $v_i \in V(G)$ **do**
  `// keyword bit vectors`
2 **for** *keyword group* $x = 1$ *to* m **do**
3 hash the keywords in the x-th keyword group of $v_i.W$ into a bit vector $v_i.BV^{(x)}$ of size B
4 **for** *each vertex* $v_i \in V(G)$ **do**
  `// neighbor keyword bit vectors`
5 initialize neighbor keyword bit vectors $v_i.NBV^{(x)}$ with $\vec{0}$ (for $1 \leq x \leq m$)
6 **for** *each neighbor vertex* $v_l \in N(v_i)$ **do**
7 **for** *keyword group* $x = 1$ *to* m **do**
8 $v_i.NBV^{(x)} = v_i.NBV^{(x)} \vee v_l.BV^{(x)}$
  `// the number of distinct neighbor keywords`
9 $v_i.nk = |\cup_{\forall v_l \in N(v_i)} v_l.W|$
  `// obtain the auxiliary data structure` $v_i.Aux$
10 $v_i.Aux = \left(v_i.BV^{(x)}, v_i.NBV^{(x)}, v_i.nk\right)$
11 **return** $v_i.Aux$

ND Lower Bound Pruning for Individual Vertices: Note that, Lemma 2 uses AND lower bound, $lb\_AND(q,g)$, to prune the entire candidate subgraphs g (which are not available during the filtering phase). Therefore, in the sequel, we will discuss how to utilize ND lower bounds, $lb\_ND(q_j,v_i)$ (w.r.t. individual vertices v_i), to filter out false alarms of vertices v_i (or retrieve candidate vertices v_i), for different aggregation functions $f(\cdot)$ (e.g., *MAX* or *SUM*).

ND Lower Bound Pruning on Individual Vertices. From Lemma 2 and Eq. (5), we can derive that: a candidate vertex v_i can be safely pruned (for either *MAX* or *SUM*), if its ND lower bound $lb\_ND(q_j,v_i)$ is greater than the aggregate threshold σ. Formally, we have the corollary below.

Corollary 4.1. **(ND Lower Bound Pruning)** *Given a query vertex $q_j \in V(q)$ and an aggregate threshold σ, a vertex v_i can be safely pruned, if it holds that $lb\_ND(q_j,v_i) > \sigma$.*

5 OFFLINE PRE-COMPUTATION

5.1 Offline Pre-Computed Auxiliary Data

To facilitate efficient online S^3AND computation, Algorithm 2 offline pre-computes relevant aggregation information for each vertex in graph G, which can be used for pruning candidate vertices/subgraphs during the S^3AND query processing. Due to space limitations, please refer to our technical report [37] for the detailed descriptions of Algorithm 2.

- **m keyword bit vectors, $v_i.BV^{(x)}$ (for $1 \leq x \leq m$), of size B,** which is obtained by using a hashing function $f(w)$ to hash each keyword $w \in v_i.W$ of each group to an integer between $[0, B-1]$ and set the $f(w)$-th bit position to 1 (i.e., $v_i.BV^{(x)}[f(w)] = 1$; lines 2-3 of Algorithm 2);
- **m neighbor keyword bit vectors, $v_i.NBV^{(x)}$ (for $1 \leq x \leq m$),** which is computed by aggregating all keywords in

$v_l.W$ from neighbor vertices $v_l \in N(v_i)$ (i.e., $v_i.NBV^{(x)} = \vee_{\forall v_l \in N(v_i)} v_l.BV^{(x)}$; lines 4-8 of Algorithm 2), and;

- **the number, $v_i.nk$, of distinct neighbor keywords,** which is given by counting the number of distinct keywords from neighbors $v_l \in N(v_i)$ of vertex v_i (i.e., $v_i.nk = |\cup_{\forall v_l \in N(v_i)} v_l.W|$; line 9 of Algorithm 2).

5.2 Indexing Mechanism

In this subsection, we show the offline construction of a tree index $\mathcal{I}$ on data graph G with the pre-computed auxiliary data in detail to support online S^3AND query computation.

The Data Structure of Index $\mathcal{I}$: We will construct a tree index $\mathcal{I}$ on the data graph G. Specifically, the tree index $\mathcal{I}$ contains two types of nodes, leaf and non-leaf nodes.

Leaf Nodes: Each leaf node $\mathcal{N}$ contains a vertex set in the data graph G. Each vertex v_i is associated with the following pre-computed data in $v_i.Aux$:

- m keyword bit vectors, $v_i.BV^{(x)}$ (for $1 \le x \le m$);
- m neighbor keyword bit vectors, $v_i.NBV^{(x)}$ (for $1 \le x \le m$), and;
- the number of distinct neighbor keywords, $v_i.nk$.

Non-Leaf Nodes: Each non-leaf node $\mathcal{N}$ has multiple entries $\mathcal{N}_i$, each corresponding to a subgraph of G. Each entry $\mathcal{N}_i$ is associated with the following aggregates:

- a pointer to a child node, $\mathcal{N}_i.ptr$;
- m aggregated keyword bit vectors, $\mathcal{N}_i.BV^{(x)} = \vee_{\forall v_l \in \mathcal{N}_i} v_l.BV^{(x)}$ (for $1 \le x \le m$);
- m aggregated neighbor keyword bit vectors, $\mathcal{N}_i.NBV^{(x)} = \vee_{\forall v_l \in \mathcal{N}_i} v_l.NBV^{(x)}$ (for $1 \le x \le m$), and;
- the maximum number of distinct neighbor keywords for vertices v_l under entry $\mathcal{N}_i$, that is, $\mathcal{N}_i.nk = \max_{\forall v_l \in \mathcal{N}_i} v_l.nk$.

Index Construction: Algorithm 3 illustrates the pseudo code of constructing a balanced tree index $\mathcal{I}$ in a top-down manner.

Specifically, given the fanout, $fanout$, of index nodes, we first calculate the tree height $h = \lceil log_{fanout}(|V(G)|) \rceil$, and use the tree root $root(\mathcal{I})$ to represent the entire vertex set $V(G)$ (lines 1-2). Then, we construct the index $\mathcal{I}$ in a top-down fashion (i.e., level l from h to 1). In particular, for each node $\mathcal{N}^{(l)}$ on the l-th level of index $\mathcal{I}$, we invoke a cost-model-based partitioning function, CM_Partitioning($\mathcal{N}^{(l)}, fanout$), to obtain $fanout$ partitions $\mathcal{N}_i^{(l-1)}$ ($1 \le i \le fanout$) of similar sizes as child nodes (lines 3-5). After we partition each non-leaf node on level $l = 1$, we obtain leaf nodes on level 0 and complete the construction of index $\mathcal{I}$. Finally, we return this balanced index $\mathcal{I}$ (line 6).

Example 4. ***(The Construction of a Tree Index, $\mathcal{I}$)*** *Figure 4 illustrates a tree index $\mathcal{I}$ over a data graph G in the example of Figure 1, via cost-model-based graph partitioning (i.e., Algorithm 4), where $n = 4$ and $\gamma = 0.2$. In particular, the tree root $\mathcal{N}_0$ (i.e., graph G) contains 4 leaf entries (i.e., 4 subgraph partitions, $\mathcal{N}_1$, $\mathcal{N}_2$, $\mathcal{N}_3$, and $\mathcal{N}_4$, respectively). Each entry (e.g., $\mathcal{N}_1$) is associated with a branch pointer $\mathcal{N}_1.ptr$ and some aggregates (i.e., $\mathcal{N}_1.BV^{(x)}$, $\mathcal{N}_1.NBV^{(x)}$, and $\mathcal{N}_1.nk$).*

Similarly, each leaf node (e.g., $\mathcal{N}_1$) contains a set of vertices (e.g., v_1 and v_{12}), where each vertex, say v_1, is associated with auxiliary data, for example, $v_1.Aux = (v_1.BV^{(x)}, v_1.NBV^{(x)}, v_1.nk)$.

Algorithm 3: The Balanced Tree Index Construction

Input: i) the pre-computed auxiliary data $v_i.Aux$ over a data graph G, and ii) the fanout, $fanout$, of the index node

Output: a balanced tree index, $\mathcal{I}$, of data graph G

1 tree height $h = \lceil log_{fanout}(|V(G)|) \rceil$
2 tree root $root(\mathcal{I}) = V(G)$
3 **for** *level $l = h$ to 1* **do**
4 **for** *each node $\mathcal{N}^{(l)}$ on the l-th level of index $\mathcal{I}$* **do**
  // cost-model-based top-down partitioning
5 invoke function **CM_Partitioning**($\mathcal{N}^{(l)}, fanout$) to obtain $fanout$ partitions $\mathcal{N}_i^{(l-1)}$ ($1 \le i \le fanout$) of similar sizes as child nodes
6 **return** $\mathcal{I}$

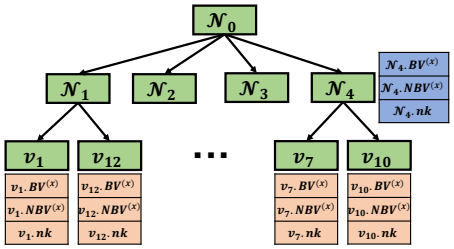


Figure 4: An example of constructing a tree index $\mathcal{I}$ based on Figure 1 (n=4, γ=0.2).

The Cost Model for Vertex Partitioning: In line 5 of Algorithm 3, we need to divide a set of vertices into n partitions of similar sizes, by invoking function CM_Partitioning $(\cdot, \cdot)$. Since different partitioning strategies may result in different pruning effects, our goal is to propose a formal cost model to guide such partitioning. Intuitively, we would like to group those vertices with similar keyword bit vectors in the same partitions (achieving high pruning power), and dissimilar bit vectors in different partitions. We use a cost model, $Cost(Par)$, to evaluate the "goodness" (quality) of the partitioning strategy Par as follows.

$$Cost(Par) = \frac{\overbrace{\sum_{i=1}^{n} \sum_{v \in Par_i} \sum_{x=1}^{m} dist(v.BV^{(x)}, c_i.BV^{(x)})}^{intra-partition\ distance\ \downarrow}}{\underbrace{\sum_{1 \le a < b \le n} \sum_{x=1}^{m} dist(c_a.BV^{(x)}, c_b.BV^{(x)}) + 1}_{inter-partition\ distance\ \uparrow}}, \quad (10)$$

where c_i is the center of partition Par_i (i.e., mean of all bit vectors in this partition), and $dist(v.BV^{(x)}, c_i.BV^{(x)})$ is given by the L_1-*norm distance* [24] between vectors $v.BV^{(x)}$ and $c_i.BV^{(x)}$ (note: in the special case of bit vectors, this is the *Hamming distance* [22]).

Here, we have:

$$dist(v.BV^{(x)}, c_i.BV^{(x)}) = \sum_{e=1}^{B} \left| v.BV^{(x)}[e] - c_i.BV^{(x)}[e] \right|, \quad (11)$$

where $v.BV^{(x)}[e]$ is the e-th position in vector $v.BV^{(x)}$, and $\Phi(\cdot)$ is an indicator function ($\Phi(z) = 1$, if z is *true*; $\Phi(z) = 0$, otherwise).

Algorithm 4: CM_Partitioning

Input: i) a set, Par, of vertices (or an index node) to be partitioned, ii) the number, n, of partitions (or the fanout of the index node), iii) the number, $global\_iter$, of global iterations, and iv) the number, $local\_iter$, of local iterations

Output: a set, $global\_Par$, of n partitions

1 $global\_cost = +\infty$;
2 **for** $k = 1$ *to* $global\_iter$ **do**
3 randomly select n center vertices $C = \{c_1, c_2, \cdots, c_n\}$, and assign each vertex v in Par to a partition, Par_i, with the closest distance $\sum_{x=1}^{m} dist(v.BV^{(x)}, c_i.BV^{(x)})$
4 obtain an initial partitioning strategy $local\_Par = \{Par_1, Par_2, \cdots, Par_n\}$ with cost $local\_cost = Cost(local\_Par)$ // Eq. (10)
5 **for** $j = 1$ *to* $local\_iter$ **do**
// update n center vertices
6 **for** $i = 1$ *to* n **do**
7 $c_i.BV^{(x)} = (\sum_{v \in Par_i} v.BV^{(x)})/|Par_i|$ (for $1 \le x \le m$)
8 **for** *each vertex* $v \in Par$ **do**
9 assign v to a partition Par'_i ($|Par'_i| \le (1+\gamma) \cdot |Par|/n$) with the closest distance $\sum_{x=1}^{m} dist(v.BV^{(x)}, c_i.BV^{(x)})$
10 obtain a new partitioning strategy $local\_Par' = \{Par'_1, Par'_2, \cdots, Par'_n\}$ with new cost $local\_cost' = Cost(local\_Par')$ // Eq. (10)
11 **if** $local\_cost' < local\_cost$ **then**
12 $local\_Par \leftarrow local\_Par'$, $local\_cost \leftarrow local\_cost'$
13 **if** $local\_cost < global\_cost$ **then**
14 $global\_Par \leftarrow local\_Par$, $global\_cost \leftarrow local\_cost$
15 **return** $global\_Par$

Intuitively, the lower value of the cost $Cost(Par)$ indicates the good quality of the partitioning strategy (i.e., with small intra-partition distances and large inter-partition distances). Thus, we aim to find a good vertex partitioning approach that minimizes the cost function $Cost(Par)$.

Cost-Model-Based Partitioning Function, CM_Partitioning $(\cdot,\cdot)$**:** Algorithm 4 illustrates the partitioning process based on our proposed cost model. Specifically, to avoid local optimality, we run our partitioning algorithm for $global\_iter$ iterations to achieve the lowest cost $global\_cost$ (lines 1-14). For each iteration, we start with a set of n random initial center vertices $C = \{c_1, c_2, \cdots, c_n\}$ and assign each vertex v in Par to a partition Par_i with the closest center vertex c_i (for $1 \le i \le n$; line 3). This way, we can obtain an initial partitioning strategy $local\_Par$ with the cost $local\_cost = Cost(local\_Par)$ (as given in Eq. (10); line 4).

Then, we will iteratively update center vertices c_i and in turn their corresponding partitions Par_i, by minimizing intra-partition distances and maximizing inter-partition distances (in light of our cost model in Eq. (10); lines 5-12). In particular, for $local\_iter$ iterations, we update the bit vectors, $c_i.BV^{(x)}$, of center vertices c_i (for $1 \le x \le m$), by taking the mean of bit vectors for all vertices in each partition Par_i (lines 6-7). We then assign vertices $v \in Par$ to a partition Par'_i with the distance $\sum_{x=1}^{m} dist(v.BV^{(x)}, c_i.BV^{(x)})$ closest to the updated centers c_i, satisfying the constraint of the balanced partitions (i.e., $|Par'_i| \le (1+\gamma) \cdot |Par|/n$), where γ is a relaxation coefficient for the partition size (lines 8-9). As a result, we obtain a new partitioning strategy, $local\_Par'$, with cost $local\_cost' = Cost(local\_Par')$ (line 10). If this new partitioning $local\_Par'$ has lower cost (i.e., $local\_cost' < local\_cost$), we will accept this new partitioning strategy by letting $local\_Par = local\_Par'$ with the lower cost $local\_cost = local\_cost'$ (lines 11-12). After $local\_iter$ iterations, we will update $global\_Par$ and $global\_cost$ with the best partitioning strategy so far and its cost, respectively (lines 13-14). Finally, we return the best partitioning strategy $global\_Par$ (after $global\_iter$ iterations) to obtain n good-quality partitions (line 15).

Time Complexity Analysis: The time complexity of offline pre-computations is given by $O(|V(G)| \cdot m \cdot (deg + \lceil log_{fanout}|V(G)| \rceil \cdot fanout \cdot local\_iter \cdot global\_iter) + \sum_{i=1}^{|V(G)|} |v_i.W|)$. Due to space limitations, please refer to our technical report [37] for the detailed complexity analysis.

6 ONLINE S^3 AND QUERY COMPUTATION

6.1 Pruning for Index Nodes

In this subsection, we propose effective pruning methods on index nodes to prune index nodes with (a group of) vertex false alarms.

Index-Level Keyword Set Pruning: If all vertices under an index entry $\mathcal{N}_i$ do not contain some keyword in $q_j.W$ for a query vertex $q_j \in V(q)$, then index entry $\mathcal{N}_i$ can be pruned w.r.t. this query vertex q_j (i.e., $\mathcal{N}_i$ does not contain any vertices matching with q_j).

Below, we provide an effective *index-level keyword set pruning* method, using the m aggregated keyword bit vectors $\mathcal{N}_i.BV^{(x)}$ stored in $\mathcal{N}_i$.

Lemma 3. **(Index-Level Keyword Set Pruning)** *Given an index entry $\mathcal{N}_i$ and a query vertex $q_j \in V(q)$, index entry $\mathcal{N}_i$ can be pruned with respect to q_j, if $\bigvee_{x=1}^{m} \left(\mathcal{N}_i.BV^{(x)} \wedge q_j.BV^{(x)} \neq q_j.BV^{(x)} \right)$ holds.*

Index-Level ND-Lower-Bound-Based Pruning: We also propose an index-level candidate node retrieval based on ND lower bounds (via Corollary 4.1) below.

Lemma 4. **(Index-Level ND-Lower-Bound-Based Pruning)** *Given an index entry $\mathcal{N}_i$, a query vertex q_j, and an aggregate threshold σ, index entry $\mathcal{N}_i$ can be safely pruned with respect to q_j, if $lb\_ND(q_j, \mathcal{N}_i) > \sigma$ holds, where we have $lb\_ND(q_j, \mathcal{N}_i) = \min_{\forall v \in \mathcal{N}_i} \{lb\_ND(q_j, v)\}$.*

Example 5. ***(Pruning Over Index I)*** *We continue with the example in Figures 1 and 4 to illustrate the index pruning in Lemmas 3 and 4. For the index-level keyword set pruning, in Figure 1, the keyword sets of vertices v_7 and v_{10} have no intersection with that of any query vertex in query graph q. Since node $\mathcal{N}_4$ contains vertices v_7 and v_{10}, its aggregates $\mathcal{N}_4.BV^{(x)}$ satisfy the condition that: $\bigvee_{x=1}^{m} \left(\mathcal{N}_4.BV^{(x)} \wedge q_j.BV^{(x)} \neq q_j.BV^{(x)} \right)$ is true (assuming no conflicts in bit vectors), for each query vertex $q_j \in V(q)$. Thus, we can safely prune entry $\mathcal{N}_4$ in root $\mathcal{N}_0$ without accessing this branch.*

For the index-level ND-lower-bound-based pruning, in Figure 4, the ND lower bound $lb\_ND(q_1, \mathcal{N}_1)$ for query vertex q_1 and node $\mathcal{N}_1$ is given by $\min_{\forall v \in \mathcal{N}_1} \{lb\_ND(q_1, v)\} = \min\{lb\_ND(q_1, v_1), lb\_ND(q_1, v_{12})$. From Eqs. (7) and 9, we have the tight ND lower

bounds $lb\_ND(q_1, v_1) = 1$ *and* $lb\_ND(q_1, v_{12}) = 2$. *Thus, we obtain* $lb\_ND(q_1, \mathcal{N}_1) = \min\{1, 2\} = 1$. *Based on index-level ND-lower-bound-based pruning (Lemma 4), if the ND lower bound,* $lb\_ND(q_1, \mathcal{N}_1)$ *(*$= 1$*), is greater than the threshold* σ*, we can safely prune entry* $\mathcal{N}_1$ *in root* $\mathcal{N}_0$ *without accessing the leftmost branch via pointer* $\mathcal{N}_1.ptr$.

6.2 S^{3}AND Query Algorithm

Algorithm 5 illustrates our proposed S^{3}AND query answering algorithm, which traverses the index $\mathcal{I}$ to retrieve candidate vertices (via pruning strategies) and refines candidate subgraphs by combining candidate vertices to return actual S^{3}AND answers.

Initialization: When an S^{3}AND query arrives, we first initialize an empty S^{3}AND query answer set S (containing subgraphs that satisfy both keyword and AND constraints w.r.t. query graph q; line 1). Moreover, for each query vertex $q_j \in V(q)$, we hash keywords in $q_j.W$ into m keyword bit vectors $q_j.BV^{(x)}$, and initialize an empty set $q_j.V_{cand}$ to record candidate vertices that match with q_j (lines 2-4). We also maintain a *maximum heap* $\mathcal{H}$ for the index traversal, accepting entries in the form $(\mathcal{N}, key)$, where $\mathcal{N}$ is an index entry, and key is a heap entry key (defined as the $\mathcal{N}.nk$; intuitively, node entries with large keys tend to contain vertices with lower ND values; line 5). Then, we add the tree root $root(\mathcal{I})$ (in the form $(root(\mathcal{I}), 0)$) to $\mathcal{H}$, and let its corresponding query vertex set $root(\mathcal{I}).Q$ be $V(q)$ (lines 6-7).

Index Traversal: We next traverse the index $\mathcal{I}$, by utilizing the maximum heap $\mathcal{H}$. Each time, we pop out an entry $(\mathcal{N}, key)$ with the maximum key from $\mathcal{H}$ (lines 8-9). When $\mathcal{N}$ is a leaf node, we will check each vertex $v_i \in \mathcal{N}$. That is, with respect to each query vertex $q_j \in \mathcal{N}.Q$, if vertex v_i cannot be pruned by the *Keyword Set* and *ND Lower Bound Pruning* (Lemma 1 and Corollary 4.1, respectively), then we add v_i to candidate vertex set $q_j.V_{cand}$ of q_j (lines 10-14).

When $\mathcal{N}$ is a non-leaf node, we consider each node entry $\mathcal{N}_i \in \mathcal{N}$ and check whether we need to access the children of entry $\mathcal{N}_i$ (lines 15-22). In particular, we first initialize an empty query set $\mathcal{N}_i.Q$, and then check if entry $\mathcal{N}_i$ can be pruned with respect to each query vertex $q_j \in \mathcal{N}.Q$ by Lemmas 3 and 4. If $\mathcal{N}_i$ cannot be ruled out (w.r.t. q_j), we will add q_j to $\mathcal{N}_i.Q$ (lines 17-20). In the case that $\mathcal{N}_i.Q$ is not empty, we insert entry $(\mathcal{N}_i, \mathcal{N}_i.nk)$ into heap $\mathcal{H}$ for later investigation (lines 21-22).

Candidate Subgraph Retrieval and Refinement: After the index traversal, we obtain a candidate vertex set $q_j.V_{cand}$ for each query vertex q_j. Since we used keyword bit vectors for pruning, there may still exist some false positives. We thus need to refine candidate vertices v_i in $q_j.V_{cand}$, by comparing the actual keyword sets (i.e., checking $q_j.W \subseteq v_i.W$; line 23).

Then, we will compute a query plan Q, which is a sorted list of query vertices in q to guide the order of candidate vertex concatenation and obtain candidate subgraphs (lines 24-27). Specifically, we initialize the first vertex in Q with a query vertex q_j with the smallest candidate set size $|q_j.V_{cand}|$ (line 24). Next, each time we append a query vertex $q_l \in V(q)$ to the end of the sorted list Q, until $Q = V(q)$ holds, where q_l is a neighbor of q_i (for all $q_i \in Q$) with the minimum candidate set size $|q_l.V_{cand}|$ (lines 25-27). After that, we call function **Refinement**$_f$ $(G, q, Q, S, \emptyset, 0, \sigma)$ in Algorithm 6, and return final S^{3}AND subgraph answers in S (lines 28-29).

Algorithm 5: Online S^{3}AND Query Processing

Input: i) a data graph G, ii) a query graph q, iii) an aggregate threshold σ, iv) an aggregation function $f(\cdot)$, and v) the index $\mathcal{I}$ over G
Output: a set, S, of S^{3}AND subgraph answers similar to q

```
// initialization
1  S ← ∅;
2  for each query vertex q_j ∈ V(q) do
3      obtain m keyword bit vectors q_j.BV^(x) (for 1 ≤ x ≤ m)
4      q_j.V_cand ← ∅;
5  initialize a maximum heap H accepting entries in the form (N, key)
6  insert entry (root(I), 0) into heap H
7  root(I).Q = V(q);
   // index traversal
8  while H is not empty do
9      (N, key) ← H.pop()
10     if N is a leaf node then
11         for each vertex v_i ∈ N do
12             for each query vertex q_j ∈ N.Q do
13                 if v_i cannot be pruned by Lemma 1 and Corollary 4.1 then
14                     q_j.V_cand ← q_j.V_cand ∪ {v_i}
15     else
           // N is a non-leaf node
16         for each entry N_i ∈ N do
17             N_i.Q ← ∅;
18             for each query vertex q_j ∈ N.Q do
19                 if N_i cannot be pruned (w.r.t., q_j) by Lemmas 3 and 4 then
20                     N_i.Q ← N_i.Q ∪ {q_j}
21             if N_i.Q is not empty then
22                 insert entry (N_i, N_i.nk) into heap H
23 refine candidate vertex sets q_j.V_cand by checking the keyword matching with q_j.W in query vertices q_j (for 1 ≤ j ≤ |V(q)|)
   // generate a query plan Q
24 obtain the first query vertex q_j with the smallest candidate vertex set |q_j.V_cand|, and initialize a sorted list (query plan) Q = {q_j}
25 while Q ≠ V(q) do
26     for all query vertices q_i ∈ Q, find a neighbor q_l ∈ N(q_i) with the minimum candidate set size |q_l.V_cand|
27     append q_l to the end of the sorted list Q
   // candidate subgraph retrieval and refinement
28 S ← Refinement_f (G, q, Q, S, ∅, 0, σ);
29 return S
```

Discussions on How to Retrieve and Refine Candidate Subgraphs: Algorithm 6 illustrates the pseudo code of a recursive function to retrieve and refine candidate subgraphs. Specifically,

Algorithm 6: Refinement$_f$

Input: i) a data graph G, ii) a query graph q, iii) a sorted candidate vertex list (query plan) Q, iv) a vertex list, M, matching with query vertices in Q, v) a recursion depth dep, and vi) an aggregate threshold σ

Output: a set, S, of subgraphs that satisfy keyword and AND constraints for q

1 **if** $|Q| = dep$ **then**
2 **if** *subgraph* g *with vertices* $V(g) = M$ *is connected and* $AND(q, g) \leq \sigma$ **then**
3 $S \leftarrow S \cup \{g\}$
4 **else**
5 **for** *each candidate vertex* $v \in Q[dep].V_{cand}$ *and* $v \notin M$ **do**
6 **if** *vertex* v *is connected to some vertex* $M[i]$ *(for* $0 \leq i < dep$*) or some candidate vertex in* $Q[i].V_{cand}$ *(for* $dep + 1 \leq i < |Q|$*)* **then**
7 $M[dep] = v$
8 **Refinement**$_f$ $(G, q, Q, S, M, dep + 1, \sigma)$
9 **return** S

in the base case that the recursive depth dep is $|Q|$, we have all the matching pairs of vertices between M and Q, and check the S^{3}AND constraints between subgraph g (with vertices in M) and query graph q. If candidate subgraph g is the S^{3}AND query answer, then we add g to the answer set S (lines 1-3).

When the recursive depth dep has not reached $|Q|$, we will consider each candidate vertex v in $Q[dep].V_{cand}$ (not a duplicate in M; line 5). In particular, if candidate vertex v is not connected to some vertex $M[i]$ $(0 \leq i < dep)$ in the current vertex list M and any vertex in $Q[i].V_{cand}$ $(dep + 1 \leq i < |Q|)$, then it implies that the resulting subgraph g will not be connected and we can terminate the recursive call; otherwise, we can set the matching vertex $M[dep]$ to v, and recursively invoke function **Refinement**$_f(G, q, Q, S, M, dep + 1, \sigma)$ for the next depth $(dep+1)$ (lines 6-8). Finally, we return the S^{3}AND query answer set S (line 9).

Complexity Analysis: The time complexity of online S^{3}AND query processing (i.e., Algorithm 5) is given by $O\big(\sum_{j=1}^{|V(q)|} |q_j.W| + \sum_{i=1}^{\lceil log_{fanout}|V(G)| \rceil} fanout \cdot (1 - PP_i) + |V(q)| + \prod_{i=1}^{|Q|} |Q[i].V_{cand}|\big)$. Due to space limitations, please refer to our technical report [37] for the detailed discussions on the complexity.

7 EXPERIMENTAL EVALUATION

7.1 Experimental Settings

Real-World Graph Data Sets: We use 5 real-world graphs, *Facebook* [38], *PubMed* [26], *Elliptic* [36], *TWeibo* [38], and *DBLPv*14 [31], whose statistics are depicted in Table 2, where "Abbr." stands for the abbreviation of the name, and "$|\sum|$" stands for the keyword domain size. *Facebook* is a social network, where two users are connected if they are friends, and keywords of each user are obtained from one's profile. *PubMed* is a citation network of scientific publications on diabetes, where keywords are from lexical features. *Elliptic* is a Bitcoin transaction network, where each node represents a transaction, edges represent financial connections, and each node contains transaction attribute category keywords. *TWeibo* is also a social network, where each node represents a user, each edge represents a following relationship, and the keyword for each node is from the user profile. *DBLPv*14 is a citation network extracted from DBLP, where each author's keywords were extracted from their relevant paper titles. It is worth noting that if a vertex in the graph mentioned above does not have keyword attributes, we create a dummy keyword label "0" to indicate *None* for this vertex.

Table 2: Statistics of the Tested Real-World Graph Data Sets.

Name	Abbr.	$\lvert V(G)\rvert$	$\lvert E(G)\rvert$	$\lvert\sum\rvert$
Facebook [38]	FB	4,039	88,234	1,284
PubMed [26]	PM	19,717	44,338	501
Elliptic [36]	EL	203,769	234,355	166
TWeibo [38]	TW	2,320,895	9,840,066	1,658
*DBLPv*14 [31]	DB	2,956,012	29,560,025	7,990,611

Table 3: Parameter Settings.

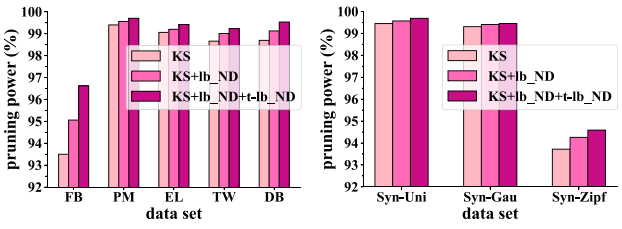
Parameters	Values
the threshold, σ_{MAX} (= σ), of MAX neighbor difference	**1**, 2, 3, 4
the threshold, σ_{SUM} (= σ), of SUM neighbor difference	2, **3**, 4, 5
the size, $\lvert v_i.W\rvert$, of keywords per vertex	1, 2, **3**, 4, 5
the size, $\lvert\sum\rvert$, of the keyword domain	10, 20, **50**, 80
the size, $\lvert V(q)\rvert$, of query graph q	3, **5**, 8, 10
the size, $\lvert V(G)\rvert$, of data graph G	10K, 25K, **50K**, 100K, 250K, 1M, 10M, 30M

Synthetic Graph Data Sets: We generate synthetic small-world graphs in the Newman-Watts-Strogatz model [35], using NetworkX [12], where parameters are depicted in Table 3. By using different distributions of keywords in $\sum$ (i.e., *Uniform*, *Gaussian*, and *Zipf*), we obtain three types of synthetic graphs: *Syn-Uni*, *Syn-Gau*, and *Syn-Zipf*, respectively.

Query Graph: For each graph data set G, we randomly sample 100 connected subgraphs. For each extracted subgraph g, we remove each of its edges with probability 0.3 (as long as the subgraph g is connected after the edge removal). As a result, we obtain 100 query graphs q for the S^{3}AND query evaluation.

Competitors: To our best knowledge, no prior work studied the S^{3}AND problem. Thus, we will compare our S^{3}AND approach (i.e., Algorithm 5) with three methods, a straightforward baseline method (named *Baseline*), *CSI_GED* [10], and *MCSPLIT* [25]. Specifically, for each vertex in a query graph q, the *Baseline* method obtains a set of candidate vertices in the data graph that match with keywords in q. Then, we aggregate those subgraphs from $|V(q)|$ candidate vertex sets that meet the S^{3}AND query requirements as the returned results. *CSI_GED* and *MCSPLIT* first retrieve a set of candidate subgraphs that are similar to a given query graph q, where the graph similarity is measured by the *Graph Edit Distance* (GED) or *Maximum Common Subgraph* (MCS), respectively. After obtaining *CSI_GED* (or *MCSPLIT*) candidate subgraphs, we next refine/return subgraphs (in these candidates) so that they satisfy the keyword set constraints and have the same size as q (note: the subgraph connectivity is relaxed and not required). In the experiments, we set the parameters to default values, and let the GED (or MCS) threshold be 1 for *CSI_GED* (or *MCSPLIT*) by default.

Measures: We evaluate the S^{3}AND query performance, in terms of *pruning power* and *wall clock time*. The *pruning power* is the percentage of candidate vertices pruned by our pruning strategies, whereas the *wall clock time* is the average time cost to answer S^{3}AND queries. We report the average values of the evaluated metrics over 100 runs.



(a) real-world graphs (b) synthetic graphs

Figure 5: Evaluation of the S^{3}AND effectiveness.

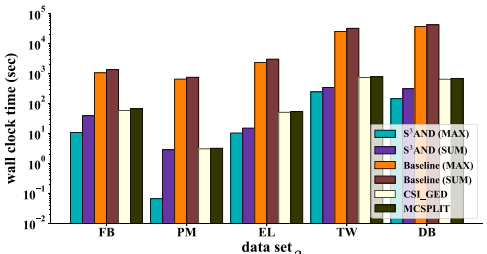


Figure 6: The comparison of our S^{3}AND approach with the *Baseline*, *CSI_GED*, and *MCSPLIT* methods over real graphs.

Parameter Settings: Table 3 shows the parameter settings, where default values are in bold. Each time we vary one parameter, while setting other parameters to default values. By default, we set the keyword group number m to 5, and the fanout, $fanout$, of index $\mathcal{I}$ to 16. For the index construction, *global_iter* and *local_iter* are set to 5 and 20, respectively. We ran all the experiments on a PC with an AMD Ryzen Threadripper 3990X CPU, 256 GB of memory. All algorithms were implemented in Python 3.11.

7.2 The S^{3}AND Effectiveness Evaluation

In this subsection, we report the pruning power of our proposed pruning strategies in Section 6.1 for S^{3}AND query processing over real-world/synthetic graphs.

Figure 5 conducts an ablation study on the pruning power of different pruning combinations in our S^{3}AND approach over real-world/synthetic graphs. Each time we add one more pruning strategy and test three pruning combinations: (1) *keyword set pruning* (*KS*), (2) *keyword set + ND lower bound pruning* ($KS + lb\_ND$; note: $lb\_ND(\cdot)$ is given by Eq. (7)), (3) *keyword set + ND lower bound pruning + Tighter ND lower bound pruning* ($KS + lb\_ND + t\text{-}lb\_ND$; note: $t\text{-}lb\_ND(\cdot)$ is given by Eq. (9)), where all parameters are set to default values. From the figures, we can see that our proposed pruning combinations can achieve high pruning powers (i.e., above 93.5%) for both real and synthetic graphs. With more pruning strategies used, our proposed S^{3}AND approach can have higher pruning power. The overall pruning power with all the three pruning methods can reach 96.62% $\sim$ 99.70% for real-world graphs and 94.59% $\sim$ 99.69% for synthetic graphs, which confirms the effectiveness of our proposed pruning strategies.

7.3 The S^{3}AND Efficiency Evaluation

In this subsection, we compare our online S^{3}AND query algorithm with *Baseline*, *CSI_GED*, and *MCSPLIT*, under default parameters over real-world graphs, in terms of the wall clock time. Figure 6 illustrates the comparative results over real-world graphs, where parameters are set to their default values. From the figure, we can see that the S^{3}AND query efficiency outperforms that of *Baseline* by 1-3 orders of magnitude, for either SUM or MAX aggregate. For example, when f is MAX, our S^{3}AND query time is 0.07 $\sim$ 246.32 *sec* for real-world graphs and 1.03 $\sim$ 5.94 *sec* for synthetic graphs. Moreover, our S^{3}AND algorithm incurs lower time cost than *CSI_GED* and *MCSPLIT* methods, due to the costly calculation of GED or MCS. Note that, since *CSI_GED* and *MCSPLIT* have different semantics from S^{3}AND and are used as the filter to retrieve candidate subgraphs, *CSI_GED* and *MCSPLIT* may not return answers of good quality (please see the case study in Figure 10 of Section 7.5).

To verify the robustness of our S^{3}AND approach, in the sequel, we will compare with *Baseline* (which also outputs exact S^{3}AND answers) and test different parameters (e.g., σ, $|v_i.W|$, $|\sum|$, $|V(q)|$, and $|V(G)|$) on synthetic graph, *Syn-Uni*. The experimental results over *Syn-Gau* and *Syn-Zipf* are similar and reported in technical report [37] due to space limitations.

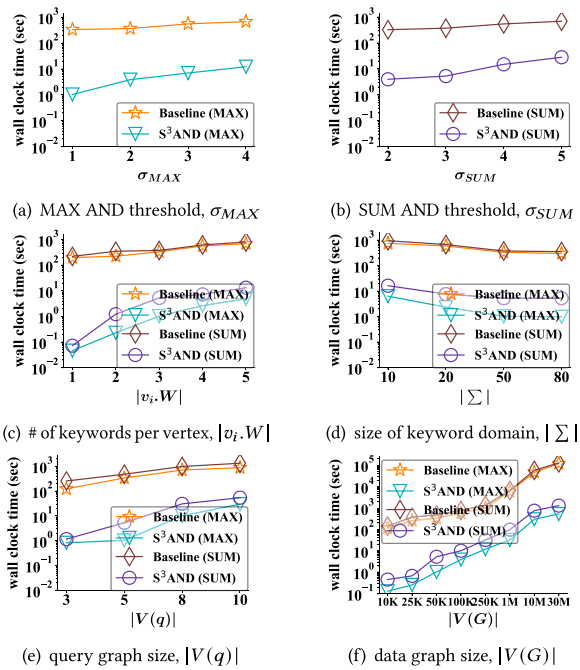
The Efficiency w.r.t. the Threshold, σ_{MAX}, of MAX Neighbor Difference: Figure 7(a) illustrates the S^{3}AND query performance for MAX aggregate (i.e., $f = MAX$), compared with *Baseline*, where the AND threshold σ_{MAX} varies from 1 to 4, and other parameters are set to default values. From the figure, we can see that for both S^{3}AND and *Baseline*, the wall clock time increases for larger σ_{MAX} over all three synthetic graphs. This is because a larger MAX threshold σ_{MAX} results in more candidate vertices, thereby raising the refinement cost. Nevertheless, our S^{3}AND approach outperforms *Baseline* by 1-3 orders of magnitude, and remains low (i.e., 1.03 $\sim$ 12.24 *sec*) over synthetic graph *Syn-Uni*.

The Efficiency w.r.t. the Threshold, σ_{SUM}, of SUM Neighbor Difference: Figure 7(b) compares the S^{3}AND query performance for SUM aggregate (i.e., $f = SUM$) with that of *Baseline*, where $\sigma_{SUM} = 2, 3, 4$, and 5, and other parameters are by default. Similar to MAX aggregate threshold σ_{MAX}, when σ_{SUM} increases, more candidate vertices will be retrieved for the refinement, which leads to higher query processing cost. Nonetheless, our S^{3}AND approach takes 3.99 $\sim$ 28.36 *sec* query time, and performs significantly better than *Baseline* by about 2 orders of maganitude.

The Efficiency w.r.t. the Number, $|v_i.W|$, of Keywords Per Vertex: Figure 7(c) reports the effect of the number, $|v_i.W|$, of keywords per vertex on the S^{3}AND query performance, where $|v_i.W|$ varies from 1 to 5, and default values are used for other parameters. With more keywords in $v_i.W$ per vertex v_i, the pruning powers of *keyword set* and *ND lower bound pruning* become lower (i.e., with more candidate vertices), which thus leads to higher time cost. Nonetheless, the S^{3}AND query cost remains low (i.e., 0.05 $\sim$ 13.23 *sec*) for different $|v_i.W|$ values, and outperforms *Baseline* by 1-3 orders of magnitude.

The Efficiency w.r.t. the Size, $|\sum|$, of the Keyword Domain $\sum$: Figure 7(d) presents the S^{3}AND query performance with $|\sum|$ = 10, 30, 50, and 80, where other parameters are set to default values. With the same number, $|v_i.W|$, of keywords per vertex, higher $|\sum|$ value incurs more scattered keywords in the keyword domain, enhancing keyword set pruning power, with fewer candidate vertices. Thus, as confirmed by the figure, for larger $|\sum|$ value, the S^{3}AND query cost decreases and remains low (i.e., 0.98$\sim$16.37 *sec*), outperforming *Baseline* by 2-3 orders of magnitude.

The Efficiency w.r.t. the Size, $|V(q)|$, of Query Graph q: Figure 7(e) demonstrates the S^{3}AND query performance for different query graph sizes $|V(q)|$, where $|V(q)|$ = 3,5,8 and 10, and other



(a) MAX AND threshold, σ_{MAX} (b) SUM AND threshold, σ_{SUM}

(c) # of keywords per vertex, $|v_i.W|$ (d) size of keyword domain, $|\sum|$

(e) query graph size, $|V(q)|$ (f) data graph size, $|V(G)|$

Figure 7: The S^{3}AND query efficiency on synthetic graph, *Syn-Uni*, compared with the *Baseline* method.

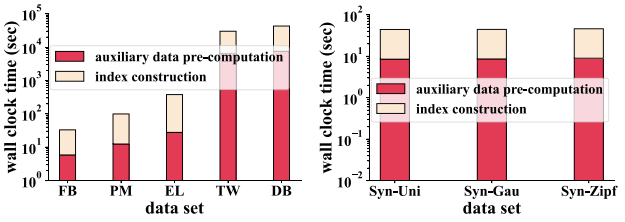
parameters are set to their default values. When the query graph size, $|V(q)|$, becomes larger, more sets of candidate vertices w.r.t. query vertices need to be retrieved and refined, resulting in higher query costs. Nevertheless, the time cost of our S^{3}AND approach still remains low (i.e., $0.82 \sim 53.91 sec$), which outperforms *Baseline* by 1-2 orders of magnitude.

The Efficiency w.r.t. the Size, $|V(G)|$, of Data Graph G: Figure 7(f) tests the scalability of our S^{3}AND approach for different data graph sizes $|V(G)|$ varying from $10K$ to $30M$, where default values are used for other parameters. From the figure, we can see that, with the increase of the data graph size $|V(G)|$, the number of candidate vertices also increases, which leads to higher retrieval/refinement costs and in turn larger query time. For large-scale *Syn-Uni* graph with $30M$ vertices, the time cost is $1,296.82$ *sec*, outperforming *Baseline* by 2-3 orders of magnitude, which confirms the efficiency and scalability of our S^{3}AND approach.

7.4 Evaluation of the S^{3}AND Offline Pre-Computations

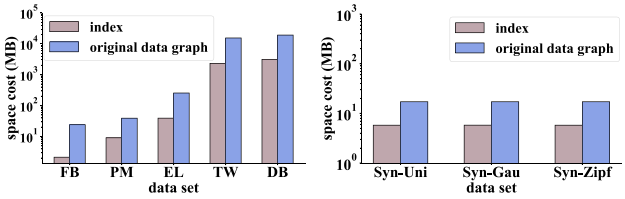
Figure 8 presents the S^{3}AND offline pre-computation cost (including time costs of auxiliary data pre-computation and index construction) over real-world/synthetic graphs, where parameters are set to default values. In Figure 8(a), for real-world graph size from $4K$ to $2.95M$, the overall offline pre-computation time varies from 33.03 *sec* to 11.85 *h*. On the other hand, for synthetic graphs, when the graph size $|V(G)|$ is $50K$, the overall offline pre-computation time in Figure 8(b) varies from 44.47 *sec* to 46.15 *sec*.

Figures 9(a) and 9(b) show the space consumption statistics for the precomputed indexes on real and synthetic graphs. From figures, we can see that for most real/synthetic graphs, the space cost of the index for our S^{3}AND algorithm is about one order of magnitude less than that of the original data graph.



(a) real-world graphs (b) synthetic graphs

Figure 8: The time costs of offline auxiliary data pre-computation and index construction.



(a) real-world graphs (b) synthetic graphs

Figure 9: The space cost of the precomputed index.

7.5 Case Study

To illustrate the effectiveness of our S^{3}AND semantics, we conduct a case study in Figure 10 and evaluate top-1 query answer of our S^{3}AND semantics (with the smallest AND score), compared with baselines *CSI_GED* and *MCSPLIT* (with the minimum GED or maximum MCS, respectively), over the *DBLPv14* graph, *DB*, where the query graph size $|V(q)|$ is set to 5.

In this case study, a user may want to conduct new multidisciplinary research, especially in quantum and 3D videos. Thus, he/she can search for a collaboration team from the DBLP graph, whose members have quantum and/or 3D videos background and have co-authored papers before. Figure 10(a) shows the targeted collaboration team, whereas Figure 10(b) provides a subgraph answer satisfying the S^{3}AND constraints (under MAX aggregate). In particular, author "Dong" is an expert in systems and quantum (i.e., matching with "q_3"), and co-authored with "Elanor" (matching with "q_1"). Other authors, *Michael*, *Mark*, and *John*, have the expertise related to the quantum and 3D videos, which include all the query keywords in each of query vertices q_2, q_4, and q_5, respectively.

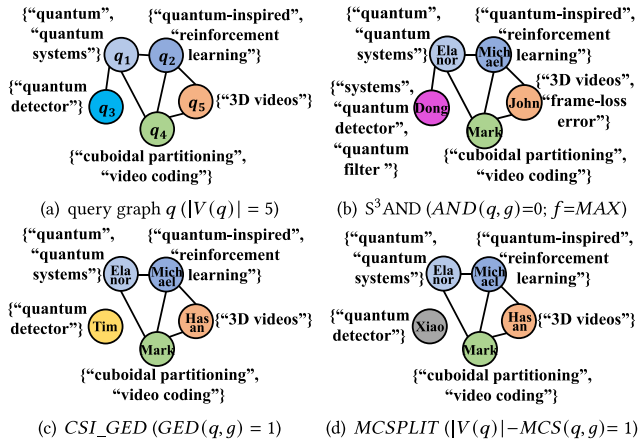
In contrast, Figures 10(c) and 10(d) return top-1 query result of *CSI_GED* and *MCSPLIT*, respectively. However, these returned subgraph answers contain isolated vertices (e.g., "Tim" in Figure 10(c) and "Xiao" in Figure 10(d)), which are not the desired collaboration teams (as some authors lack co-author relationships with others, and may incur high communication/cross-learning costs). Thus, our S^{3}AND query more effectively returns subgraphs satisfying both keyword and neighbor difference conditions in this scenario.

We also tested other parameters (n, $global\_iter$, $local\_iter$, σ_{MAX}, and σ_{SUM}), and other graph data (*Syn-Gau* and *Syn-Zipf*). Full results and parameter tuning details are in our technical report [37].

8 RELATED WORK

8.1 Subgraph Matching

Exact Subgraph Matching: Existing works on exact subgraph matching considered backtracking-search-based [3, 4, 13] and multi-way-join-based algorithms [17, 18, 30]. Backtracking-search-based



(a) query graph q ($|V(q)| = 5$) (b) S^{3}AND ($AND(q,g)$=0; f=MAX)

(c) CSI_GED ($GED(q,g) = 1$) (d) $MCSPLIT$ ($|V(q)| - MCS(q,g) = 1$)

Figure 10: A case study of S^{3}AND, *CSI_GED*, and *MCSPLIT* over *DB* graph data with query graph size 5.

algorithms perform deep matching of a given query graph via vertex-to-vertex mapping and backtrack upon failure. CECI [3], CFLMatch [4], DP-iso [13] optimize the overhead of generating intermediate results, by using preprocessing-enumeration paradigms. An example of multi-way-join-based algorithm is GpSM [32], which merges candidate edges based on pairwise join to obtain matching results and is suitable for tree-shaped or non-cyclic graph queries. Recently, embedding or learning-based approaches such as GNN-PE [40] employed path embeddings for the exact subgraph matching problem over a data graph under the graph isomorphism semantics, where path embeddings are defined as the concatenation of embedding vectors from vertices on the path (i.e., embeddings of these vertices and their 1-hop neighbors), learned by *Graph Neural Networks* (GNNs). GNN-PE assumed that each vertex in the data graph is only associated with a single keyword (rather than a keyword set in S^{3}AND), and the subgraph matching considers graph isomorphism (instead of S^{3}AND matching semantics such as keyword set containment and aggregated neighbor difference constraints). Therefore, with a different graph data model and query semantics, we cannot directly use previous techniques in GNN-PE for tackling our S^{3}AND problem.

Approximate Subgraph Matching: When the response time is more critical than the accuracy, approximate subgraph matching can efficiently return top-k approximate subgraphs that are similar to the query graph, and is widely used in real applications [11, 14, 23]. Existing approximate subgraph matching algorithms usually searched for top-k similar subgraphs from a (large) data graph, by setting different similarity metrics for various scenarios, e.g., GED [10, 15, 42] and GBD [20]. Although these matches can give answers quickly, they do not guarantee the accuracy and are more limited to the task scenario (e.g., the algorithms cannot give the exact locations of similarity subgraphs in the data graph).

8.2 Subgraph Similarity Search

Previous works on subgraph similarity search extensively studied the subgraph partitioning [21, 43], filtering optimization [6, 39], and indexing retrieval [16, 33, 34] to improve the efficiency. NeMa [16] obtained top-k subgraphs with minimum matching costs, defined as the sum of *keyword matching* and *distance proximity costs* between query and data vertices. The *keyword matching cost* is given by the *Jaccard similarity* over keyword sets from a pair of query and data vertices. Moreover, the *distance proximity cost* is defined as the difference between *neighborhood vectors* [16] from a pair of query and data vertices, where the *neighborhood vector* records distances from a vertex to its h-hop neighbors. In contrast, our S^{3}AND query semantics consider the containment relationship of keyword sets for the vertex matching (i.e., different from the *Jaccard similarity measure* in NeMa), and take into account the structural difference between subgraph g and query graph q (i.e., aggregated 1-hop neighbor difference of each vertex v_i, compared with query vertex q_j) which differs from the NeMa semantics (i.e., the distance proximity cost, caring more about the similarity of distances from query/data vertices to their h-hop neighbors). SLQ [39] obtains the top-k subgraphs with the highest ranking scores, given by the sum of *edge* and *node matching costs*, where the *edge matching cost* (or *node matching cost*) is defined as the (weighted) number of transformation functions (pre-defined in a library) that can transform the data edge (or data node) to the query edge (or query node). Different from SLQ that considered the graph data model with semantic information in vertices and edges, the graph model in our S^{3}AND problem assumes vertices associated with keyword sets. Furthermore, our S^{3}AND query semantics focus on the 1-hop neighbor structural difference between subgraph g and query graph q, which differs from the *ranking scores* in SLQ. Thus, with different graph data models and query semantics, we cannot directly apply the approaches proposed in NeMa and SLQ to tackle our S^{3}AND problem. Recently, many embedding-based subgraph similarity search algorithms [1, 2, 19, 27, 41] have been proposed, which incur faster online processing time. However, the accuracy and model training cost of these methods are still insufficient for the needs of critical applications. Due to different graph similarity semantics, we cannot directly borrow previous techniques to solve our S^{3}AND problem.

9 CONCLUSIONS

In this paper, we formulate a novel problem, *subgraph similarity search under aggregated neighbor difference semantics* (S^{3}AND), which has broad applications (e.g., collaborative team detection and fraud syndicate identification) in real-world scenarios. To enable efficient S^{3}AND queries, we propose effective pruning strategies to rule out false alarms of candidate vertices/subgraphs. We devise an index over offline pre-computed data, which can help apply our pruning methods to retrieve candidate subgraphs during the index traversal. Finally, we conduct extensive experiments to confirm the performance of our S^{3}AND approach on real/synthetic graphs.

10 ACKNOWLEDGEMENTS

This work was supported by Natural Science Foundation of China (62272170), Natural Science Foundation (NSF CCF-2217104), and Shanghai Trusted Industry Internet Software Collaborative Innovation Center. Qi Wen, Yutong Ye, and Mingsong Chen are with the MoE Engineering Research Center of SW/HW Co-Design Technology and Application, East China Normal University. Mingsong Chen is the corresponding author.

REFERENCES

[1] Yunsheng Bai, Hao Ding, Song Bian, Ting Chen, Yizhou Sun, and Wei Wang. 2019. Simgnn: A neural network approach to fast graph similarity computation. In *Proceedings of the twelfth ACM international conference on web search and data mining*. 384–392.

[2] Franka Bause, Erich Schubert, and Nils M Kriege. 2022. EmbAssi: embedding assignment costs for similarity search in large graph databases. *Data Mining and Knowledge Discovery* 36, 5 (2022), 1728–1755.

[3] Bibek Bhattarai, Hang Liu, and H. Howie Huang. 2019. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. ACM, 1447–1462.

[4] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient Subgraph Matching by Postponing Cartesian Products. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. ACM, 1199–1214.

[5] David B Blumenthal, Nicolas Boria, Johann Gamper, Sébastien Bougleux, and Luc Brun. 2020. Comparing heuristics for graph edit distance computation. *The VLDB journal* 29, 1 (2020), 419–458.

[6] Xiaoyang Chen, Hongwei Huo, Jun Huan, and Jeffrey Scott Vitter. 2019. An efficient algorithm for graph edit distance computation. *Knowledge-Based Systems* 163 (2019), 762–775.

[7] Sourav Dutta, Pratik Nayek, and Arnab Bhattacharya. 2017. Neighbor-aware search for approximate labeled graph matching using the chi-square statistics. In *Proceedings of the 26th International Conference on World Wide Web*. 1281–1290.

[8] Xinbo Gao, Bing Xiao, Dacheng Tao, and Xuelong Li. 2010. A survey of graph edit distance. *Pattern Analysis and applications* 13 (2010), 113–129.

[9] Jessica Gliozzo, Alex Patak, Antonio Puertas Gallardo, Elena Casiraghi, and Giorgio Valentini. 2023. Patient Similarity Networks Integration for Partial Multimodal Datasets. SCITEPRESS, 228–234.

[10] Karam Gouda and Mosab Hassaan. 2016. CSI_GED: An efficient approach for graph edit similarity computation. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 265–276.

[11] Shunan Guo, Zhuochen Jin, Qing Chen, David Gotz, Hongyuan Zha, and Nan Cao. 2022. Interpretable Anomaly Detection in Event Sequences via Sequence Matching and Visual Comparison. *IEEE Trans. Vis. Comput. Graph.* 28, 12 (2022), 4531–4545.

[12] Aric Hagberg, Pieter J Swart, and Daniel A Schult. 2008. *Exploring network structure, dynamics, and function using NetworkX*. Technical Report. Los Alamos National Laboratory (LANL), Los Alamos, NM (United States).

[13] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. ACM, 1429–1446.

[14] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *Proceedings of the 2019 International Conference on Management of Data*. 1429–1446.

[15] Rashid Ibragimov, Maximilian Malek, Jiong Guo, and Jan Baumbach. 2013. Gedevo: an evolutionary graph edit distance algorithm for biological network alignment. In *German conference on bioinformatics 2013*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[16] Arijit Khan, Yinghui Wu, Charu C Aggarwal, and Xifeng Yan. 2013. Nema: Fast graph search with label similarity. *Proceedings of the VLDB Endowment* 6, 3 (2013), 181–192.

[17] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, et al. 2019. Distributed subgraph matching on timely dataflow. *Proceedings of the VLDB Endowment* 12, 10 (2019), 1099–1112.

[18] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, Ying Zhang, Zhengping Qian, and Jingren Zhou. 2019. Distributed Subgraph Matching on Timely Dataflow. *Proc. VLDB Endow.* 12, 10 (2019), 1099–1112.

[19] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. 2019. Graph matching networks for learning the similarity of graph structured objects. In *International conference on machine learning*. PMLR, 3835–3845.

[20] Zijian Li, Xun Jian, Xiang Lian, and Lei Chen. 2018. An efficient probabilistic approach for graph similarity search. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 533–544.

[21] Yongjiang Liang and Peixiang Zhao. 2017. Similarity search in graph databases: A multi-layered indexing approach. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 783–794.

[22] Alex X Liu, Ke Shen, and Eric Torng. 2011. Large scale hamming distance query processing. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 553–564.

[23] Xiaoxiao Ma, Jia Wu, Jian Yang, and Quan Z. Sheng. 2023. Towards Graph-level Anomaly Detection via Deep Evolutionary Mapping. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2023, Long Beach, CA, USA, August 6-10, 2023*. ACM, 1631–1642.

[24] MD Malkauthekar. 2013. Analysis of euclidean distance and manhattan distance measure in face recognition. In *Third International Conference on Computational Intelligence and Information Technology (CIIT 2013)*. IET, 503–507.

[25] Ciaran McCreesh, Patrick Prosser, and James Trimble. 2017. A partitioning algorithm for maximum common subgraph problems. (2017).

[26] Zaiqiao Meng, Shangsong Liang, Hongyan Bao, and Xiangliang Zhang. 2019. Co-embedding attributed networks. In *Proceedings of the twelfth ACM international conference on web search and data mining*. 393–401.

[27] Zongyue Qin, Yunsheng Bai, and Yizhou Sun. 2020. GHashing: Semantic graph hashing for approximate similarity search in graph databases. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*. 2062–2072.

[28] Niranjan Rai and Xiang Lian. 2023. Top-k Community Similarity Search Over Large-Scale Road Networks. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 35, 10 (2023), 10710–10721.

[29] Qi Song, Yinghui Wu, Peng Lin, Luna Xin Dong, and Hui Sun. 2018. Mining summaries for knowledge graph search. *IEEE Transactions on Knowledge and Data Engineering* 30, 10 (2018), 1887–1900.

[30] Shixuan Sun, Xibo Sun, Yulin Che, Qiong Luo, and Bingsheng He. 2020. Rapidmatch: A holistic approach to subgraph query processing. *Proceedings of the VLDB Endowment* 14, 2 (2020), 176–188.

[31] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. 2008. Arnetminer: extraction and mining of academic social networks. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. 990–998.

[32] Ha-Nguyen Tran, Jung-jae Kim, and Bingsheng He. 2015. Fast subgraph matching on large graphs using graphics processors. In *International Conference on Database Systems for Advanced Applications*. Springer, 299–315.

[33] Guoren Wang, Bin Wang, Xiaochun Yang, and Ge Yu. 2010. Efficiently indexing large sparse graphs for similarity search. *IEEE Transactions on Knowledge and Data Engineering* 24, 3 (2010), 440–451.

[34] Xiaoli Wang, Xiaofeng Ding, Anthony KH Tung, Shanshan Ying, and Hai Jin. 2012. An efficient graph indexing method. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 210–221.

[35] Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of 'small-world'networks. *nature* 393, 6684 (1998), 440–442.

[36] Mark Weber, Giacomo Domeniconi, Jie Chen, Daniel Karl I Weidele, Claudio Bellei, Tom Robinson, and Charles Leiserson. 2019. Anti-Money Laundering in Bitcoin: Experimenting with Graph Convolutional Networks for Financial Forensics. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

[37] Qi Wen, Yutong Ye, Xiang Lian, and Mingsong Chen. 2025. S^3AND: Efficient Subgraph Similarity Search Under Aggregated Neighbor Difference Semantics (Technical Report). *arXiv preprint arXiv:2505.00393* (2025).

[38] Renchi Yang, Jieming Shi, Xiaokui Xiao, Yin Yang, Sourav S Bhowmick, and Juncheng Liu. 2023. PANE: scalable and effective attributed network embedding. *The VLDB Journal* 32, 6 (2023), 1237–1262.

[39] Shengqi Yang, Yinghui Wu, Huan Sun, and Xifeng Yan. 2014. Schemaless and structureless graph querying. *Proceedings of the VLDB Endowment* 7, 7 (2014), 565–576.

[40] Yutong Ye, Xiang Lian, and Mingsong Chen. 2024. Efficient exact subgraph matching via gnn-based path dominance embedding. *Proceedings of the VLDB Endowment* 17, 7 (2024), 1628–1641.

[41] Yutong Ye, Xiang Lian, and Mingsong Chen. 2024. Efficient Exact Subgraph Matching via GNN-based Path Dominance Embedding. *Proc. VLDB Endow.* 17, 7 (2024), 1628–1641.

[42] Zhiping Zeng, Anthony KH Tung, Jianyong Wang, Jianhua Feng, and Lizhu Zhou. 2009. Comparing stars: On approximating graph edit distance. *Proceedings of the VLDB Endowment* 2, 1 (2009), 25–36.

[43] Xiang Zhao, Chuan Xiao, Xuemin Lin, Qing Liu, and Wenjie Zhang. 2013. A partition-based approach to structure similarity search. *Proceedings of the VLDB Endowment* 7, 3 (2013), 169–180.

[44] Weiguo Zheng, Lei Zou, Wei Peng, Xifeng Yan, Shaoxu Song, and Dongyan Zhao. 2016. Semantic SPARQL similarity search over RDF knowledge graphs. *Proceedings of the VLDB Endowment* 9, 11 (2016), 840–851.