Hindawi
Complexity
Volume 2021, Article ID 9274429, 18 pages
https://doi.org/10.1155/2021/9274429

Research Article

Supergraph Topology Feature Index for Personalized Interesting Subgraph Query in Large Labeled Graphs

Xiaohuan Shan, Haihai Li, Chunjie Jia, Dong Li, and Baoyan Song

College of Information, Liaoning University, Shenyang 110036, China

Correspondence should be addressed to Baoyan Song; bysong@lnu.edu.cn

Received 16 April 2021; Revised 7 May 2021; Accepted 4 June 2021; Published 29 June 2021

Academic Editor: chuan lin

Copyright © 2021 Xiaohuan Shan et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Interesting subgraph query aims to find subgraphs that are isomorphic to the given query graph from a data graph and rank the subgraphs according to their interestingness scores. However, the existing subgraph query approaches are inefficient when dealing with large-scale labeled data graph. This is caused by the following problems: (i) the existing work mainly focuses on unweighted query graphs, while ignoring the impact of query constraints on query results. (ii) Excessive number of subgraph candidates or complex joins between nodes in the subgraph candidates reduce the query efficiency. To solve these problems, this paper proposes an intelligent solution. Firstly, an Isotype Structure Graph Compression (ISGC) strategy is proposed to compress similar nodes in a graph to reduce the size of the graph and avoid unnecessary matching. Then, an auxiliary data structure Supergraph Topology Feature Index (STFIndex) is designed to replace the storage of the original data graph and improve the efficiency of an online query. After that, a partition method based on Edge Label Step Value (ELSV) is proposed to partition the index logically. In addition, a novel Top-K interest subgraph query approach is proposed, which consists of the multidimensional filtering (MDF) strategy, upper bound value (UBV) (Size-c) matching, and the optimizational join (QJ) method to filter out as many false subgraph candidates as possible to achieve fast joins. We conduct experiments on real and synthetic datasets. Experimental results show that the average performance of our approach is 1.35 higher than that of the state-of-the-art approaches when the query graph is unweighted, and the average performance of our approach is 2.88 higher than that of the state-of-the-art approaches when the query graph is weighted.

1. Introduction

In recent years, along with the growing popularity of the Internet, a mass of data with closely related to real entity has been produced [1–3], which has grown explosively [4] in various fields. Taking social networks as an example, in 2018, the monthly active users of WeChat stabilized at about 1.12 billion, and 45 billion messages were sent every day [5]. In 2019, Facebook officially announced that the total number of active users has exceeded 2.45 billion [6]. As an efficient data structure, labeled graphs [7] can better express the internal connections of various entities [8] for dealing with such large-scale data [9].

Subgraph query aims to search all matching subgraphs in a data graph that are isomorphic to the given query graph and have correct labels [9–11]. It is widely used to describe, mine, and analyze various knowledge graphs [12]. For example, in the anti-fraud knowledge graph for credit cards, we can monitor fraud gangs by defining fraud gangs (subgraph schema) and query them. By using a subgraph query, various social relations of an enterprise can be captured and analyzed from an enterprise knowledge graph. In addition, by using subgraph query, excellent teams can be mined in a military management knowledge graph, interest groups can be searched in a social knowledge graph, and target groups can be found in an e-commerce knowledge graph for a personalized recommendation.

In practice, users are more interested in a few matching subgraphs with higher-ranking scores, rather than the whole query results of a subgraph query. Usually, a subgraph with a ranking score is called an interesting subgraph, and the ranking score is calculated by the interestingness score [13]

of a subgraph, which is the sum of weights. In order to reduce the negative impact of information overload, the concept of Top-K interesting subgraph query [13] is proposed, that is, selecting K subgraphs with the highest-ranking scores from the results of a subgraph query. For example, Figure 1(a) shows a data graph G abstracted from DBLP, in which nodes A, J, and F represent author, journal, and research field, respectively; the weight on each edge denotes the degree of correlation between two entities. Figure 1(b) shows a query graph Q for the Top-K interesting subgraph query. Assuming $K = 2$, then the Top-K interesting subgraphs isomorphic to Q in the data graph G are P_1 (interestingness score is 3.4) and P_2 (interestingness score is 3.0).

Recently, various approaches were proposed for Top-K interesting subgraph query. However, most of the existing subgraph query approaches are inefficient when dealing with large-scale graphs. This is caused by the following problems: (i) the existing work mainly focuses on unweighted query graphs, while ignoring the impact of query constraints on query results. (ii) Excessive number of subgraph candidates or complex joins between nodes in the subgraph candidates reduce the query efficiency.

Specifically, in practice, users often want to get more specific and accurate query results. This can be achieved by adding constraints of filters to a query graph. Taking Figure 1(a) as an example, for the given query graph Q and data graph G, we can find all the isomorphic subgraphs of Q in G. Then, if adding constraints or filters to the query graph, we can find more accurate subgraph matching results in the query results, such as to find teams (i.e., subgraphs) with frequent cooperation between members. This can be realized by adding edge weights between two entities with label A in the query graph. In other words, it is to find the isomorphic subgraphs of the query graph with the highest-ranking scores on the data graph, and the edge weight between any two author nodes on the isomorphic subgraphs is greater than or equal to the edge weight between two author nodes on the query graph. Finding isomorphic subgraphs with edge weights greater than or equal to those on the query graph is called a constraint or filter. Thus, in this paper, the process of obtaining subgraph query results by constraining or filtering a query graph is called a personalized Top-K interesting subgraph query. It refers to the K matching subgraphs with the top-ranking scores among the whole query results that are isomorphic to the query graph and satisfying constraints or filters.

To solve the above problems, this paper proposes an intelligent solution. Specifically, an Isotype Structure Graph Compression (ISGC) strategy is first proposed to compress similar nodes in a graph to reduce the size of the graph and avoid unnecessary matching. Then, an auxiliary data structure Supergraph Topology Feature Index (STFIndex) is designed to replace the storage of the original data graph and improve the efficiency of an online query. After that, a partition method based on Edge Label Step Value (ELSV) is proposed to partition the index logically. In addition, a novel Top-K interest subgraph query approach is proposed, which consists of the multidimensional filtering (MDF) strategy, UBV (Size-c) matching, and the optimizational join (QJ) method to filter out as many false subgraph candidates as possible to achieve fast joins.

We conduct experiments on real and synthetic datasets. Experimental results show that the average performance of our approach is 1.35 higher than that of the state-of-the-art approaches when the query graph is unweighted, and the average performance of our approach is 2.88 higher than that of the state-of-the-art approaches when the query graph is weighted.

The contributions of this paper can be summarized as follows:

(i) We propose an Isotype Structure Graph Compression (ISGC) strategy. Specifically, in a data graph, it compresses adjacent nodes that have the same label into a folding node that contains multiple raw data nodes. Then, the edges connected from a node to a folding node are compressed to form a critical compression edge. The compressed folding nodes and critical compression edges can not only reduce the size of the data graph but also filter multiple invalid nodes and edges in batch.

(ii) We design an index called Supergraph Topology Feature Index (STFIndex), which represents all topology information in the generated compressed graph. As an auxiliary data structure, it can replace the original data graph storage. STFIndex will be created offline to improve the efficiency of online queries. It consists of Node Compression Topology Index (NCTIndex) and Edge Compression Feature Index (ECFIndex), which can be used to filter false candidates of nodes and edges.

(iii) To improve the computational efficiency of subgraph query in a large labeled graph, we propose a partition method based on Edge Label Step Value (ELSV). It logically divides STFIndex into multiple partitions, enabling parallel queries for each partition.

(iv) By using STFIndex, we propose a personalized Top-K interesting subgraph query approach in large labeled graphs, called STF_ISQtop-k. It contains query graph preprocessing, multidimensional filtering (MDF) strategy, UBV (Size-c) matching, and an optimizational join (OJ) method. MDF is used to filter out as many false candidates as possible, and OJ implements the fast joins of subquery results.

Organization. The rest of this paper is organized as follows: in Section 2, we introduce the background of this paper, which contains the problem statement and the related works. ISGC strategy and STFIndex structure are discussed in Section 3. We provide the details of the proposed Top-K interesting subgraph query approach in Section 4. Experimental results and analysis are discussed in Section 5. Section 6 is the conclusion.

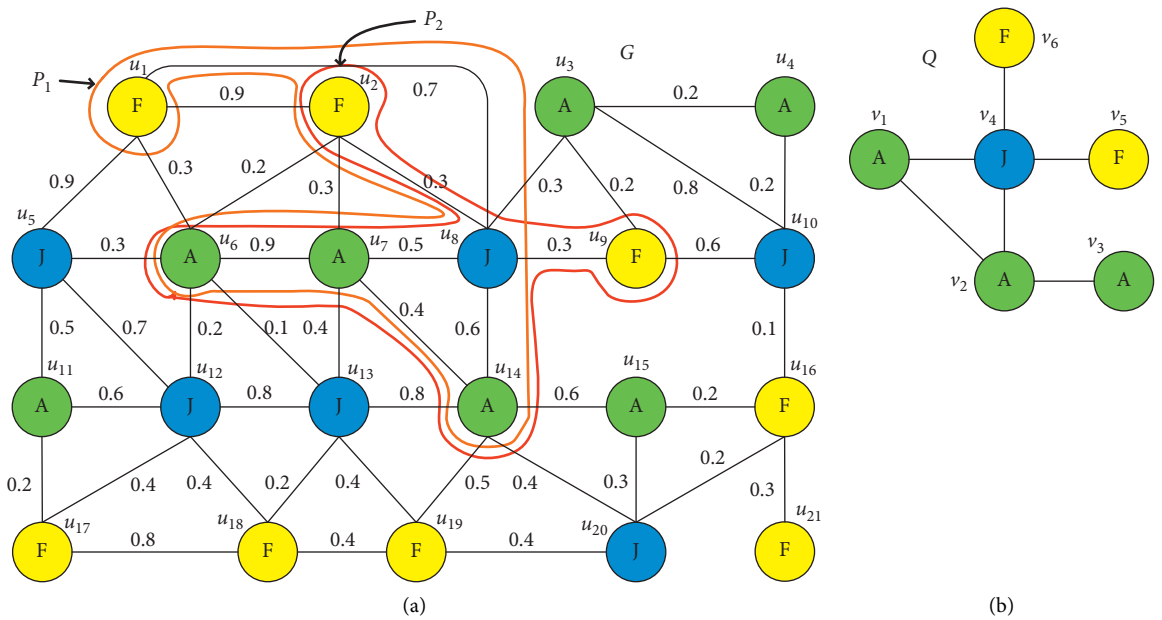

Figure 1: Data graph G in (a) and query graph Q in (b). P_1 and P_2 are the isomorphic subgraphs of the query graph Q.

2. Background

2.1. Problem Statement. A data graph can be represented as a 4-tuple $G = (V, E, L, W)$, where V denotes the set of nodes (or vertices); E ($\subseteq V \times V$) is the set of edges; L ($V \longrightarrow \Sigma$) presents a function labeling the nodes; Σ is the sets of labels that can appear on the nodes; and W is the set of edge weights. In this paper, the data graph we studied is an undirected labeled graph, in which an undirected edge between nodes u and v is denoted indifferently by (u, v) or (v, u).

Conventional subgraph query problem is usually defined as given a query graph Q and a data graph G, to find all the isomorphic subgraphs of Q in G. With the explosion of data volume and data complexity, researchers turned to research the approaches to search K optimal results by using edge weights, i.e., Top-K interesting subgraph query [13]. However, in practice, users hope to add one or more constraints of edge weights to the query graph to satisfy more accurate (or personalized) queries. This query can be called a personalized Top-K interesting subgraph query in this paper.

We use the interestingness score [14] to identify the quality of a matching subgraph, which is the sum of the weights of a matching subgraph in the data graph G. It is generally believed that the higher the interestingness score, the better the matching result. Thus, the personalized Top-K interesting subgraph query is used to search the top K matching subgraphs with the highest interestingness score under the condition that the query constraint is satisfied.

2.2. Related Work. Subgraph query has a long research history. The earliest and most classical subgraph query algorithm, named Ullmann algorithm [15], was based on state-space search, which laid a foundation for the subsequent research on subgraph query. However, the Ullmann algorithm adopted a recursive exhaustive method without defining the matching order. As the size of data graphs increases, the query efficiency of the Ullmann algorithm decreased significantly.

As early as 1979, subgraph query has been proved to be an NP-complete problem [16], and it is difficult to find the optimal results in a limited time. Thus, VF2 [17] improved the pruning strategy of the Ullmann algorithm. It proposed a set of feasibility decision rules based on the original state-space representation and then defined the matching order of query nodes. However, both Ullmann and VF2 were limited by the superlinear complexity of large-scale graphs.

Some algorithms [18–20] constructed indexes based on frequent substructures to quickly match the substructures contained in a query graph. Such approaches are efficient when the query graph is frequently structure. Since the same node or edge may exist in multiple frequent subgraphs, the improper mining of frequent structures can lead to excessive redundant storage when building indexes in a large-scale graph. In addition, if the query graph does not contain frequent substructures, only plain query approaches can be made. In other words, frequent subgraph indexes limit the generality of the algorithms.

GraphQL [21] and SPath [22] constructed indexes based on reachable neighbors, and they pruned the candidate sets by capturing the information of neighbor nodes or edges and defining the corresponding optimization strategies. GraphQL [21] used the information of neighbor nodes and breadth-first search to minimize candidate sets, which filtered out invalid nodes and edges. SPath [22] constructed the index by the shortest path of nodes within the D-hop range. The index contained the number of neighbor labels per hop and the ID of the corresponding node for each label. The former was stored in faster memory, while the latter was stored in memory or out of memory, depending on the size

of the index. GraphQL and SPath have the problems of undefined matching order and repeated enumeration of equivalent nodes.

CFLMatch [23] delayed Cartesian product operation to reduce the number of intermediate results for subgraph matching. It built a Compact Path-Index (CPI) according to the query graph in real-time to achieve query matching. CFLMatch used an adjacency matrix representation (with size $|V| \times |V|$) of the data graph, which limited that it can only be used to process small-scale data graphs. CECI [24] utilized the BFS-based filtering and reverse-BFS-based refinement to prune the unpromising candidates and then replaced the edge verification with set intersection to speed up the candidate verification. In addition, similar algorithms include the literature [25–28].

Most index-based algorithms adopted the principle of filtering the nodes in the query graph one by one, and only considered the structural relationships between the current node and its adjacent nodes. They ignored the relationships between nodes of the same label type, leading to a lot of double counting. TurboISO [29] proposed the Comb/Perm strategy, which used neighborhood equivalence classes to generate a combination for each NEC during a subgraph isomorphism search. The similar nodes in the query graph are then combined into a single supernode. Instead of arranging all possible enumerations, it only filtered the supernodes.

BoostISO [30] focused on the characteristics between nodes. It not only considered the relationships between nodes in a query graph but also extended the compression techniques to a data graph. It also merged similar nodes in the data graph. When TurboISO and BoostISO applied graph compression techniques to subgraph queries, they focused on the redundancy of similar nodes, while ignoring the redundancy of dissimilar nodes. A subgraph query algorithm based on the ASSG model [31] was proposed in [31] to compress the data graph into a graph that can adapt to the structure of the query graph. It reduced the computation in the matching process. However, for the subgraph query of weighted graphs, none of these algorithms contained a weight filtering strategy. This resulted in an inefficient late filtering process that negated the benefits of earlier queries.

SumISO framework [32] performed graph compression with modular decomposition. The compressed graph was parsed into several regions containing query graphs, using supervertex selection to find the modules that may match the query graph. Then, the backtracking algorithm [32] was used to find a successfully matching query graph in each region. Multiple iterations of the backtracking algorithm made it difficult to extend to large graphs.

The Top-K interesting subgraph query mainly includes two parts: one is to find all matching results from a data graph, which are isomorphic to the given query graph. The other is to extract the first K subgraphs according to the sum of weights on the data graph. The personalized Top-K interesting subgraph query adds the query constraints by the given weights in the query graph.

RAM [33] constructed the SPath index structure and proposed a filter-validation strategy, which can obtain all the matching subgraphs. And then it extracted the K matching results with the greatest interestingness. Since the process of obtaining all the matching results was relatively time-consuming, it was inefficient to query on large-scale data graphs. Different from RAM, RWM [13] proposed the idea of ranking with matching. The definition of Top-K interesting subgraph query was first standardized in RWM. It proposed two index structures, graph topology index and maximum metapath weight index. An optimized filtering strategy was also proposed to filter the obvious unsatisfied results. RWM did not require sorting and screening, which effectively improved the query efficiency.

A Top-K search framework STAR was proposed in [34], which consisted of two parts: a fast Top-K algorithm for star queries and an assembling algorithm for general graph queries. It was effective for processing query graphs with no weights. In [35], a new graph similarity measure was introduced based upon subtree patterns for the Top-K graph similarity search. It created hierarchical representative matrices for fast search. DSQL [36] was a level-based algorithm with an approximation guarantee, which was proposed to find K subgraphs isomorphic to a given query graph with maximum coverage. However, it was only applied to unweighted graphs. TKG [37] was proposed to find Top-K frequent subgraphs. It started searching for patterns using an internal MinSup threshold set to 0 and gradually raised the threshold as patterns were found. It had a certain limitation that it only searched the frequent subgraphs.

PBSM [38] introduced graph compression techniques into the Top-K interesting subgraph query to reduce the enumeration. In the filtering stage, it reduced the memory cost by only considering the direct neighbors. In the verification stage, it took the vertex with the minimum number of candidate vertices in the query graph as the start vertex of the matching order and used the idea of ranking while matching (RWM) to terminate the execution of the algorithm as early as possible by estimating the upper bound of the weights. This can reduce redundant verification and improve the overall performance. In [39], it proposed two types of interestingness measurements IE-Match and C-Match based on weighted query semantics. Moreover, it devised a space-efficient indexing scheme DVI to improve the indexing and candidate generation process.

CSEA pattern [40] provided to the user a set of attributes that had exceptional values throughout a subset of vertices. The strength of the proposed pattern language lied in its independence to a notion of support to assess the interestingness of a pattern. It was only applied to the attributed subgraphs without weights. W-Gaston [41] was proposed to modify the support in the Gaston algorithm. The proposed interesting approach integrated the frequency, page duration, and entropy parameters to mine frequent interesting subgraphs.

3. Data Graph Compression and STFIndex Creation

This section discusses the techniques to preprocess the data graph offline. We design an Isotype Structure Graph

Compression (ISGC) strategy to compress the connected nodes with the same labels and edges. Then, based on the generated compression graph, we create STFIndex to capture the candidates of query graphs. In order to extend the method to the large graphs, STFIndex is divided into several partitions for parallel processing based on the ELSV partition principle.

3.1. Isotype Structure Graph Compression. Structure profile is an important data compression model for XML documents at the earliest, which aims to simplify the description of XML document tree structures. The XML structure profile is also a tree structure, and it has no nodes with the same label and path. To narrow the size of the graph and speed up a query, this paper introduces the idea of XML structure profile into data graph compression and proposes an Isotype Structure Graph Compression (ISGC) method. Nodes with the same label and connected to each other are called equivalent nodes, and the edge between them is called isotype edge. Other nodes in the graph are called non-equivalent nodes, and other edges are known as heterotype edges.

ISGC aims to merge equivalent nodes of a data graph by using equivalent relations. In this paper, the basis for judging whether nodes are equivalent or not is based on the structure of the graph and the labels of nodes, that is, whether nodes have the same labels and been connected in the graph. In the process of merging equivalent nodes, the merged node is called the folding node and the internal connections in a folding node are internal edges. With the merging of nodes, edges connected between some nodes to internal nodes within the folding node are merged. The merged edge is called critical compression edge, and its weight is the maximum weight of the internal edge. To distinguish the folding node from the nonfolding node, we set the first bit of the folding node *id* at 0. We call the compressed graph at supergraph, and there are no isotype edges inside.

Figure 2 shows the example of graph compression. Thereinto, u_1 and u_2 are compressions into folding node u_{01}, and the critical compression edge is (u_{01}, u_6, 0.3). Figure 3 shows the supergraph G_C of the data graph G in Figure 1, in which the folding nodes are marked red.

3.2. STFIndex Creation. In order to improve the efficiency of invalid nodes and edges filtering, this paper presents a multilayer index structure, namely, Supergraph Topology Feature Index (STFIndex). STFIndex consists of the Node Compression Topology Index (NCTIndex) and the Edge Compression Feature Index (ECFIndex). The NCTIndex can retrieve the information of nodes to filter invalid nodes whose topological structures do not meet the constraints. The ECFIndex index can be used to filter false candidates edges based on the types of edges and weights.

3.2.1. NCTIndex. It is found that topological properties such as types of labels and information of adjacent nodes with different labels are symbolic and distinguishable in labeled

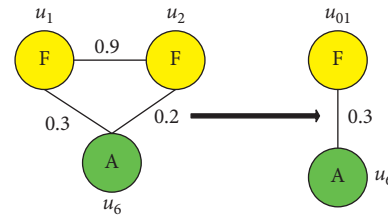

Figure 2: Example of graph compression.

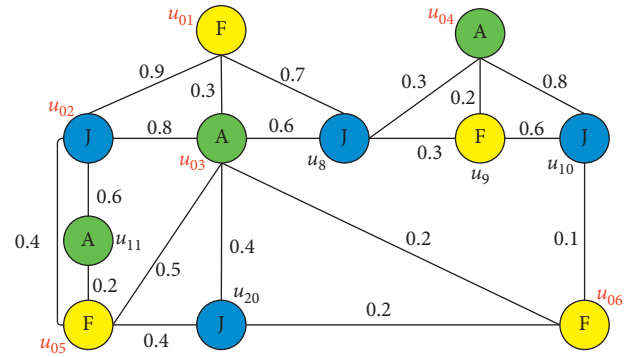

Figure 3: The supergraph of data graph G in Figure 1.

graphs. Therefore, NCTIndex is designed based on them in this paper, which is composed of three layers.

The bottom layer index is created through the topological relationship of nodes in the supergraph, which is a multituple of <*n_id*, num_{lable1}, num_{lable2}, . . ., num_{lablen}>. Thereinto, *n_id* is the *id* of each node that is the subscript of each node *u*, and num_{lable1}, num_{lable2}, . . ., num_{lablen} represent the number of adjacent nodes with each label, respectively. The middle layer index is created according to the topological relationship of nodes contained in the folding nodes of the supergraph. It is composed of a 4-tuple <*f_id*, *count*, *max-w*, *child* []>; among them, *f_id* is the *id* of the folding node that is the subscripts of the folding node *u*, *count* represents the number of nodes compressed by the folding node, *max-w* is the maximum weight of the internal edge, and the *child* [] is the pointers to the bottom layer. The top layer index is created according to the label types and the number of nodes, which consists of <label type of node, the number of nodes with the current label, and pointer to the middle layer or bottom layer>. The pointer to the bottom layer will point to the unmerged nodes.

For example, Figure 4 shows the NCTIndex of supergraph G_C in Figure 3. In the top layer, we can see that G_C has three kinds of labels A, J, and F, which contain 3, 4, and 4 nodes, respectively. In the middle layer, for label A, it contains two folding nodes, whose *id*s are 03 and 04. The folding node 03 is merged by four nodes, and the max weight is 0.9. The bottom layer stores the original nodes and the number of their adjacent nodes for different labels.

3.2.2. ECFIndex. Analyzing the characteristics of the labeled graph, it is found that not only do the nodes contain a lot of information, but also the information of edges

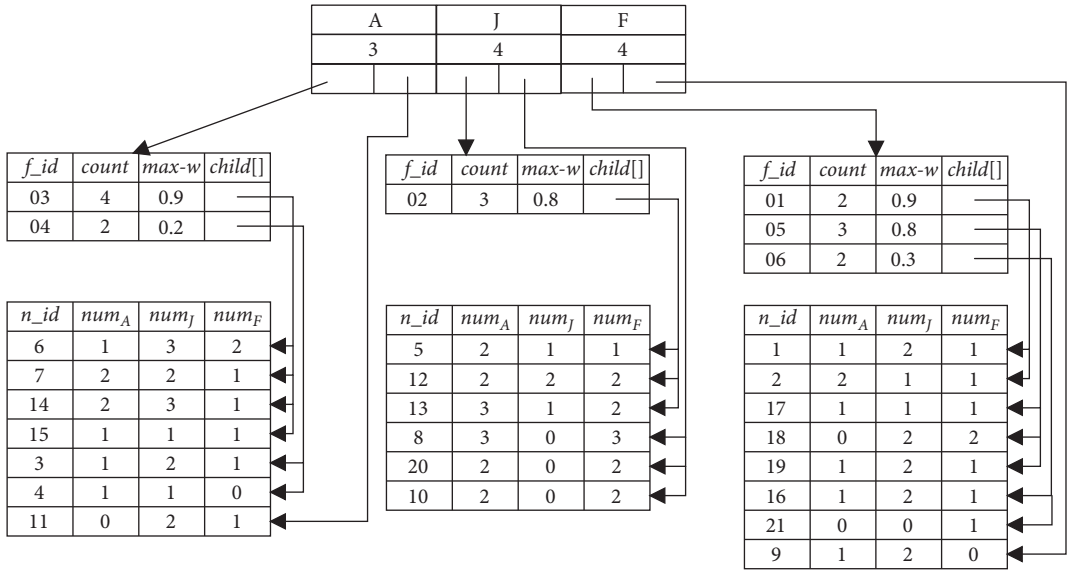

Figure 4: The NCTIndex of supergraph G_C in Figure 3.

such as edge type (represented by the combination of the node labels) and weight is useful to filter the candidates. Thus, in order to further filter the invalid structure, this paper proposes the ECFIndex. It will be divided into Isotype Edge Compression Feature Index (I-ECFIndex) and Heterotype Edge Compression Feature Index (H-ECFIndex) according to whether the label types of endpoints are the same.

I-ECFIndex contains a two-level index. The bottom layer index represents the information of edges for each kind of types, and it is composed of $<id_1, id_2, w>$, where id_1 and id_2 are *id*s of endpoints. To improve the filtering speed, the bottom layer index is stored in descending order of weights. The top layer index is created according to the label types of edges.

H-ECFIndex consists of a three-level index. The bottom layer index is used to establish the edge information contained in each label type, which is represented by a 3-tuple $<id_1, id_2, w>$. And it is also stored in descending order of weights. According to the features of compressed edges in folding nodes, the middle layer index contains the endpoints of the edge, the max weight of the folding node, and the pointer to the bottom layer. It is represented by $<id_1, id_2,$ *max-w*, *child* []>. The top layer index is created in accordance with label types of edges. As shown, Figures 5(a) and 5(b) are I-ECFIndex and H-ECFIndex of supergraph G_C in Figure 3.

Since all the information of the data graph can be read in the bottom indexes of STFIndex (consists of NCTIndex and ECFIndex), which is built created offline. To avoid redundant storage, the original data graph can be dumped or deleted as required, which reduces the memory overhead effectively. The space complexity of STFIndex is $O(|E|)$, which E is the set of edges. STFIndex incurs a higher space cost due to the replication of the information of nodes, but it accelerates the filtering process. The time complexity of STFIndex creation is $O(|E|)$.

3.3. STFIndex Partition. In order to fully improve the computing efficiency of subgraph query in a large labeled graph, we divide the STFIndex into several partitions, which partitions the original data graph indirectly, to realize the parallel query in each partition.

Firstly, we introduce the concept Edge Label Step Value (ELSV), and the standardized definition is defined as follows.

Definition 1. Edge Label Step Value (ELSV): given a group label of graph G, $L = \{L_1, L_2, \ldots, L_n\}$ (n is a positive integer), custom setting the order of labels in L, such as $L_1 > L_2 \ldots > L_n$ thereinto, ">" represents the sequential order of labels, and such $L_1 > L_2$ indicates that L_1 comes before L_2. Then, ELSV of either edge label L_iL_j (i and j are positive integers) is as follows:

$$\mathrm{ELSV}\left(L_iL_j\right) = |i - j|. \tag{1}$$

In the phase of subgraph matching, the number of edge label types contained in the data graph directly determines iterations of matching layers. In order to make matching layers distribute to each partition evenly, a partition method based on the ELSV principle is proposed. In real complex networks such as military management networks, social networks, and so on, the edge size is usually several times the node size, or even an order of magnitude higher than that. Therefore, considering the memory overhead of replication, we divide the index based on nodes, which needs to replicate partial node information of the NCTIndex rather than a huge number of edge information.

The main ideas of the ELSV partition method are to calculate the sorted ELSVs in the data graph and set a partition for each label. The NCTIndex is divided according to the label types of nodes logically. For example, if there are four labels a, b, c, and d in the data graph, then we will divide NCTIndex into four partitions, respectively, NCTIndex_a, NCTIndex_b, NCTIndex_c, and NCTIndex_d. For

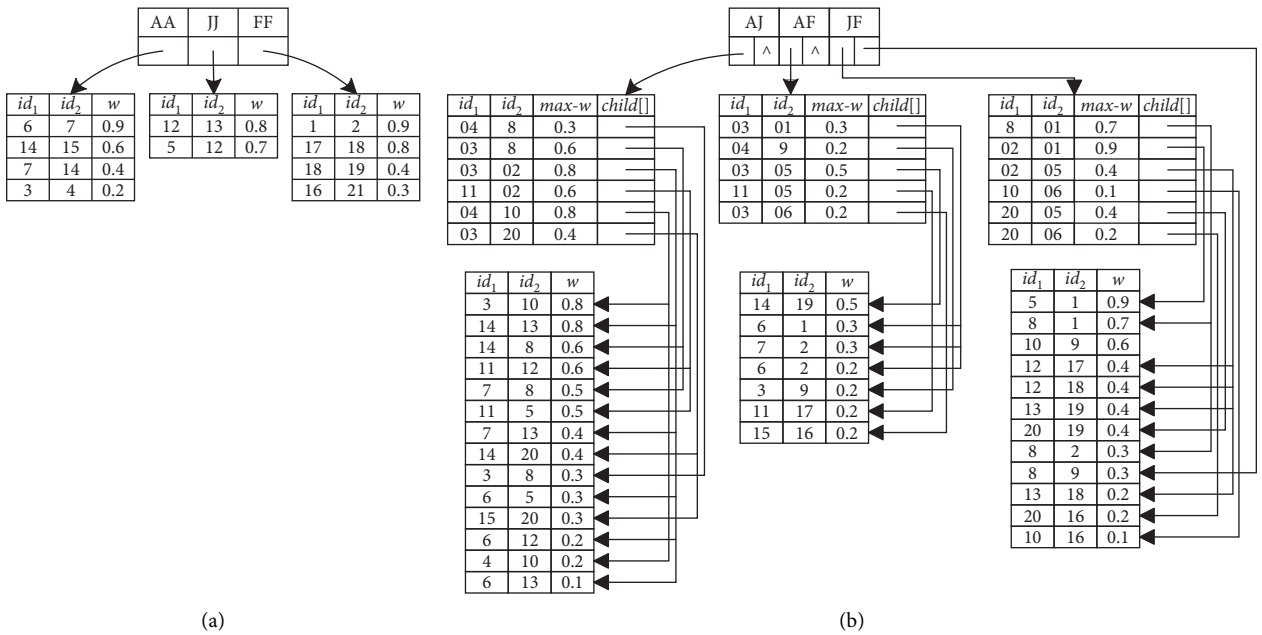

Figure 5: ECFIndex of supergraph G_C in Figure 3. It consists of I-ECFIndex in (a) and H-ECFIndex in (b).

ECFIndex, I-ECFIndex is divided into the partition that the endpoints belong. H-ECFIndex is divided vertically into partitions according to the parity of the ELSVs. If the ELSV is odd, it is stored on the partition of the endpoint with a smaller label value and if it is even, it is stored on the partition of the endpoint with a larger label value, and vice versa. Such as if the order of labels is a > b > c > d, then the ELSV of edge label ab is 1, which is odd. Thus, we store the ECFIndex_ab in partition b. In order to guarantee the correctness of the filter, each partition should replicate the information of "special" nodes from NCTIndex without needing to replicate the whole NCTIndex. The "special" nodes have the same labels as the current partition in the ECFIndex. Lemma 1 states the quantity relationship of the number of index items between NCTIndex and each partition. We exploit this relationship for verifying the effectiveness of the proposed partition method.

Lemma 1. *Let data graph be G = (V, E, L, W), then Num (NCTIndex_X) = Num (NCTIndex)/2 + 1 and X ∈ L. Num (NCTIndex_X) represents the number of labels in partition X, and Num (NCTIndex) is the number of labels in data graph G.*

Proof. According to the ELSV partition rule, assume there are N labels in the data graph that Num (*NCTIndex*) is N, and label X ranks K in the label order. The *NCTIndex_X* that is part of *NCTIndex* stored in partition X contains three categories: (1) the index items of NCTIndex with label X; (2) the index items of NCTIndex with labels sorting before K, in which the ELSV is odd and there are index items of adjacent nodes with label X; (3) the index items of NCTIndex with labels sorting after K, in which the ELSV is even and there are index items of adjacent nodes with label X. So, in case (1), there is only one kind of node label X. In case (2), the number of labels is $K/2$, and in case (3), that is $(N - K)/2$, then Num $(NCTIndex\_X) = 1 + K/2 + (N - K)/2 = N/2 + 1$. In conclusion, Num (*NCTIndex_X*) = Num (*NCTIndex*) /2 + 1.

When replicating the NCTIndex, we will filter and only replicate index items of the NCTIndex having the same label with the current partition, rather than the whole NCTIndex. Therefore, the size of the index items corresponding to each label in partition X is less than or equal to that in the original data graph. Since Num (*NCTIndex_X*) = Num (*NCTIndex*)/2 + 1, the replication quantity of per partition is much less than the whole NCTIndex. Therefore, the ELSV partition method avoids duplicate storage of ECFIndex and greatly reduces the overhead of NCTIndex index redundant storage.

In addition, the label frequency of nodes is normally power-law distributed and the edges follow a similar pattern. So, the size of one label partition may be significantly larger than the other. To manage the imbalances between partitions, we adopt the combination of the logical and physical partition. We use the proposed strategy based on ELSV to partition indexes logically, and then, the partitions are sorted by the number of nodes and edges. According to the number of machines in the cluster, we assign the high-ranked (the partition with more nodes and edges) and low-ranked partitions into the same physical partition, to solve the problem of imbalance as far as possible.

Taking Figure 3 as an example, there are three types of nodes A, J, and F and six types of edges AA, AJ, AF, JJ, JF, and FF. Thus, we set three partitions A, J, and F to store the nodes of three types in NCTIndex, respectively, NCTIndex_A, NCTIndex_J, and NCTIndex_F. We assume label order is A > J > F, and isotype edges AA, JJ, and FF are stored in partition A, J, and F, respectively. And we calculate the ELSVs of heterotype edges, which are ELSV(AJ) = 1, ELSV(AF) = 2, and ELSV(JF) = 1. ELSVs of heterotype edges with AJ and JF are odd, so the index items of AJ in ECFIndex

are stored in partition J and JF are stored in partition F. ELSVs of edges with AF are even, so the index items are stored on the partition of the endpoints with larger label values A. Above all, partition A contains index items about AA and AF of ECFIndex, partition J contains those about JJ and AJ, and index items of FF and JF are stored in partition F. Since the heterotype edge label AF is divided into partition A, the information of label A in NCTIndex_F needs to be replicated in partition A. The other heterotype edges are similar. The final partition result is shown in Table 1, $S(X_Y)$ represents the information of nodes with label Y in NCTIndex_X, and S(X) is the NCTIndex of label X.

The example (NCTIndex_J and ECFIndex_J) of partition J is as shown in Figure 6. □

4. Personalized Top-K Interesting Subgraph Query

4.1. Query Graph Preprocessing. Before the interesting subgraph query, we preprocess the query graph includes query graph compression and partition, which is executed by the same methods as the data graph. For example, the process of compression and partition of query graph Q is shown in Figure 7. First of all, nodes v_1, v_2, and v_3 have the same label A, we compress them into folding node v_{01}, and it means that the query graph Q is converted into the supergraph Q_c. Then, we construct STFIndex for Q_c with the same index items of the data graph to filter candidates by the index. Finally, in accordance with the partition principle based on ELSV, Q_c is divided into different partitions to realize the subsequent query.

4.2. Multidimensional Filter. The scale and accuracy of candidates will directly influence the efficiency of matching. Creating the effective index and filter strategy can exclude inconsistency with the query graph, which can reduce the unnecessary verification to improve the efficiency of subgraph matching.

In this paper, a multidimensional filter strategy (MDF) is proposed based on NCTIndex and ECFIndex to pruning nodes and edges. To acquire the candidate of nodes, NCTIndex is used in this paper to effectively prune the nodes that do not meet the requirements. We take into account such factors as the label type of nodes, the size of folding nodes, and the number of adjacent nodes.

Based on the acquired candidate node set (CNS), we utilize ECFIndex to excavate the information of nodes and edges. We consider the factors such as weights and the effectiveness of endpoints to prune the query graph further. Given the above filters, we can obtain candidate edge set (CES), which is the intermediate set to subsequent subgraph matching. We know most constraints of the query are limited by weights, and we subdivide edges of the query graph into common edges and special edges (weights is not zero). MDF strategy applies five filters as follows, in which (1) and (2) are used to filter the nodes and edges are filtered by (3)–(5):

Table 1: Temperature and wildlife count in the three areas covered by the study.

Partition	NCTIndex	I-ECFIndex	H-ECFIndex
Partition A	$S(A)$, $S(F_A)$	AA	AF
Partition J	$S(J)$, $S(A_J)$	JJ	AJ
Partition F	$S(F)$, $S(J_F)$	FF	JF

(1) Folding node location filter (FNLF), for a folding node, the number of compressed nodes and a maximum weight of internal edges must be no less than that of the query graph. Therefore, NCTIndex is used to prune folding nodes that do not meet the constraints and filter their bottom layer data in the index. At the same time, the nodes without compressing in NCTIndex are all filtered. This is because they are single nodes. For example, the query graph is Q_C_J of partition J in Figure 7, the index of data graph is shown in Figure 6, we find only a folding node v_{01} with label A in Q_C_J, and the middle layer of NCTIndex with label A is retrieved to acquire u_{03} and u_{04}. Since the number of compressed nodes in u_{04} is 2, which is less than the folding node v_{01} in the query graph, u_{04} and its bottom index items are filtered out, and u_{11} is filtered as a single node. The number of compressed nodes and a maximum weight of u_{03} are both more than v_{01}, so the u_{03} is the candidate node.

(2) Adjacent node label frequency filter (ANLFF), the candidate nodes should satisfy that the number of their adjacent nodes with different labels are not less than that of the query graph. Thus, based on the FNLF strategy, we utilize NCTIndex to prune the nodes, which do not satisfy the occurrence frequency of the adjacent nodes with each label. After that, we can acquire a smaller CNS. For example, we can acquire the candidate sets of v_1, v_2, v_3, and v_4 in query graph are $\{u_6, u_7, u_{14}, u_{15}\}$, $\{u_7, u_{14}\}$, $\{u_6, u_7, u_{14}, u_{15}\}$, and $\{u_{12}, u_{13}, u_8, u_{20}, u_{10}\}$.

(3) Critical compression edge filter (CCEF), for a critical compression edge, which has been introduced in 3.1 that multiple edges connected from a node to a folding node are merged, its maximum weight should not be less than the weight of the query graph. Thus, the critical compression edges whose weights do not meet the constraints are pruned by ECFIndex, and the bottom layer data have been filtered indirectly. Since the bottom of ECFIndex is sorted in descending order according to weights, we do not need to traverse over the items that are less than the constraint. Thus, filter in batches is realized. In our example, since $(v_1, v_4, 0.5)$ and $(v_2, v_4, 0)$ are compressed into critical compression edge $(v_{01}, v_4, 0.5)$, and the type of edge is AJ, we filter out edges $(u_{04}, u_8, 0.3)$ and $(u_{03}, u_{20}, 0.4)$ and their corresponding bottom index items by the middle layer of ECFIndex, whose weights dissatisfy the constraint.

(4) Bottom special edge filter (BSEF), for a special edge, we retrieve the bottom index of the ECFIndex and

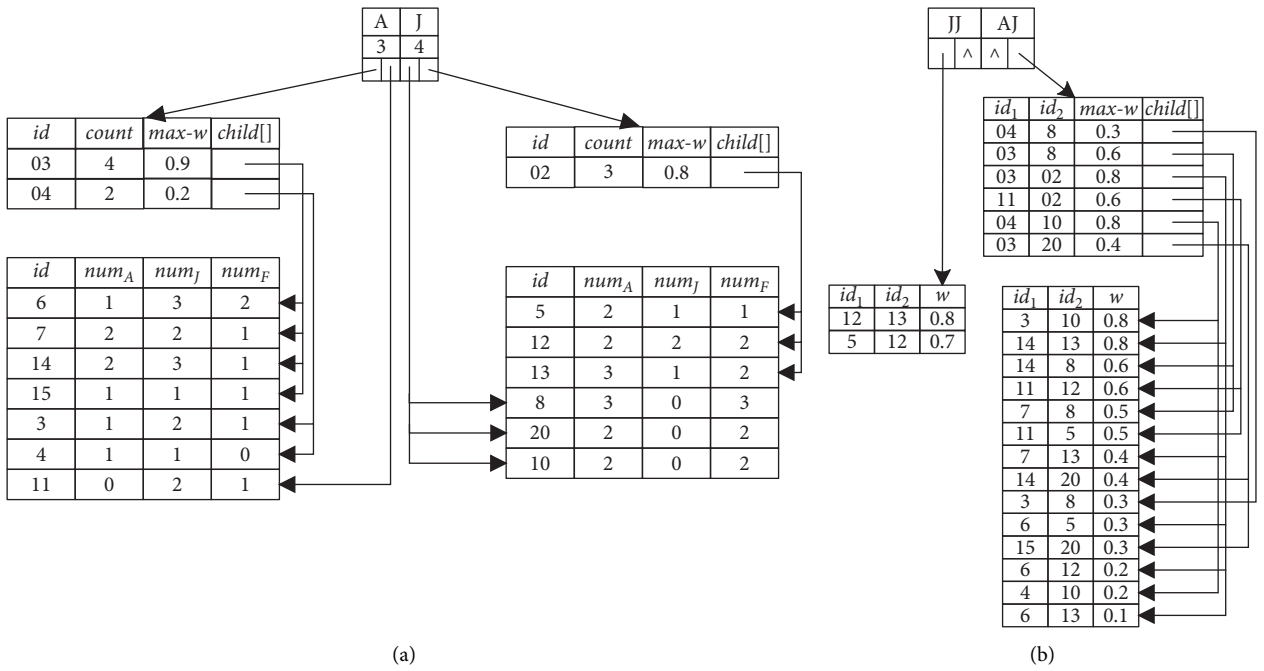

Figure 6: The NCTIndex_J in (a) and ECFIndex_J in (b) of partition J.

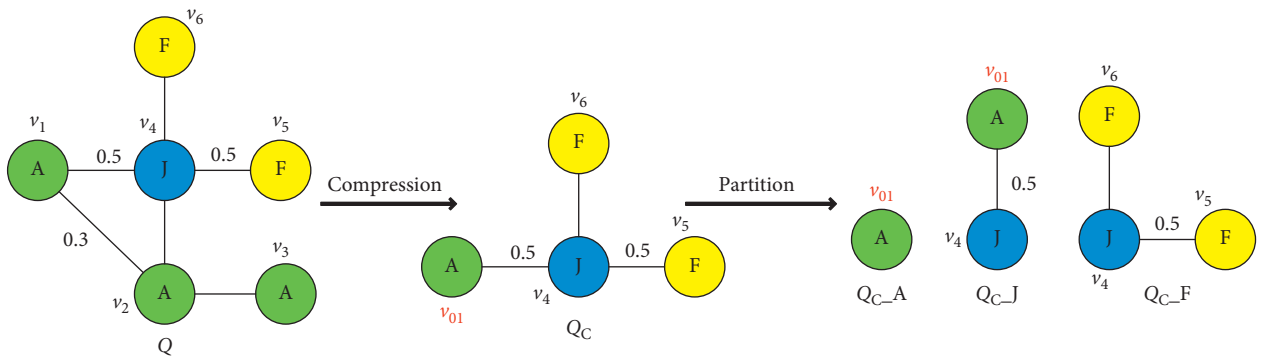

Figure 7: The compression and partition of query graph Q. Q is the query graph with edge weights, Q_C is the compressed query graph of Q, and Q_C is divided into three partitions Q_C_A, Q_C_J, and Q_C_F.

select the edges that have the same label with query graph, and weights are not less than the current special edge. Then, we prune the false candidate edges, whose endpoints are not in CNS. Thus, it can filter the CES further. Likewise, (v_1, v_4, 0.5) is a special edge with the edge label AJ. We retrieve the bottom of ECFIndex_J to get its edge candidate set {(u_{14}, u_{13}, 0.8), (u_{14}, u_8, 0.6), and (u_7, u_8, 0.5)}.

(5) Bottom common edge filter (BCEF), for a common edge, we also retrieve the bottom index of the ECFIndex based on the label to acquire the edges, which have the same label with the current common edge, and the endpoints are both in CNS. For the common edge (v_2, v_4, 0), we get its edge candidate set {(u_{14}, u_{13}, 0.8), (u_{14}, u_8, 0.6), (u_7, u_8, 0.5), and (u_7, u_{13}, 0.4)}.

4.3. UBV (Size-c) Matching. In this section, matching is performed on the candidate edge set CES. We select the minimum candidate edge set as the initial matching edge for breadth-first search (BFS) to reduce iterations and calculation overhead effectively. Before introducing the method of matching, the concepts of Size-c matching and its upper bound value (UBV) are introduced as follows.

Definition 2. Size-c matching: it represents a partial growth matching when the c edges of the query graph are instantiated in the process of subgraph matching, where $c \in (1, n)$ and n is the number of edges in the query graph. And the interestingness score is the sum of weights that have been instantiated.

Definition 3. Upper bound value (UBV): it is the upper bound value of interestingness score in Size-c matching. It is

equal to the sum of weights that both have been instantiated and uninstantiated with the maximum candidate edge weight in the process of the Size-c matching.

The specific steps of matching are shown in Algorithm 1. The algorithm maintains two heaps, Top-K heap and candidate matching (CM) heap. The Top-K heap stores K matching results in descending order. The CM heap stores the matching results in ascending order. Some matches have been inserted into the Top-K heap but may be overwritten by subsequent matching. Other matches have not been inserted into the Top-K heap but have been verified. The array CP is used to store the matching results temporarily, and $O(|E_Q|)$ is the order of matching (line 2). The starting edge and matching order are determined in lines 3-4. Query graph Q_J is instantiated to get the Top-K heap and CM heap in lines 5–15. If the interestingness score of Size-c () is equal to or less than the bottom of the Top-K heap, then the result will be stored in the CM heap (lines 7-9). Else, the bottom of the Top-K heap will be replicated to the CM heap and deleted from the Top-K heap, and at the same time, the result will be stored in the Top-K heap (lines 11–15). The time complexity of the matching algorithm is $O(n)$, in which n is the number of edges in CES.

Take the candidate edge set CES_J in Figure 8 as an example, and K is assumed 2. Firstly, we traverse CES_J to determine the initial matching edge (v_1, v_4, 0.5). Then, BFS is performed from this edge to get the matching order (v_1, v_4, 0.5) ⟶ (v_2, v_4, 0). According to the matching order, (u_{14}, u_{13}, 0.8) is instantiated (v_1, v_4, 0.5) to obtain the Size-1 candidate matching (u_{14}, v_2, u_{13}): 0.8. By that analogy, Size-c part growth matching ($c = 2 \ldots n$ and n is the number of edges in query graph Q_J) is performed to obtain K matching results (u_{14}, u_7, u_{13}): 1.2 and (u_{14}, u_7, u_8): 1.1, which are stored in Top-K heap. Continue the instantiation to get the Top-K heap and *CM_J* heap of partition J. Top-K and CM heaps of each partition are as shown in Figure 9.

4.4. Query Join. An optimizational join (OJ) algorithm is proposed to join the matching results of each partition and obtain the complete K matching subgraphs of query graph Q. It proceeds as few communications times as possible to reduce the communication cost in the join stage. Before describing the optimizational join algorithm, Lemma 2 is introduced first.

Lemma 2. *If the top K matching results have been joined and the minimal interestingness score is μ, the interestingness score of the lowest possible matching result in the Top-K heap of partition i (Top-K_i) is μ_i. Eventually, the top K matching subgraphs can be acquired by joining the matching results in each partition, whose interestingness scores are greater than or equal to $\mu_i = \mu - \sum_{j=1,j\neq i}^{n} INT(r_{j,1})$. Among them, $r_{j,1}$ represents the first result in Top-K heap of partition j, $INT(r_{j,1})$ is the interestingness score of matching result $r_{j,1}$ (since the Top-K heap is in descending order, the interestingness score of the first result is largest.), and n is the number of partitions.*

Proof. Assume that the minimum interestingness score of the final Top-K matching subgraphs is μ_k, thus $\mu_k \geq \mu$, and then $\mu_k \geq [\mu - \sum_{j=1,j\neq i}^{n} INT(r_{j,1})] + \sum_{j=1,j\neq i}^{n} INT(r_{j,1})$. Make $\mu_i = \mu - \sum_{j=1,j\neq i}^{n} INT(r_{j,1})$, and thus, $\mu_k \geq \mu_i + \sum_{j=1,j\neq i}^{n} INT(r_{j,1})$. Thus, join the matching results in the Top-K_i heap of partition i whose interestingness scores are less than μ_i with the matching results of other partitions. The interestingness score of generated matching result will be less than μ_k; thus, it is impossible to be the final matching result.

We know from Lemma 2 that if the minimum interestingness score of matching results in partition i is greater than the lower bound value $\mu_i = \mu - \sum_{j=1,j\neq i}^{n} INT(r_{j,1})$of possible matching in this partition, then it illustrates that the current partition may acquire better results by joining with other partitions. Therefore, the matching results that are not less than μ_i in partition i will be joined with other partitions to update complete Top-K results.

In the OJ algorithm, the query subgraph contains most nodes in the initial matching, because it is easier to instantiate. The query order is obtained by BFS of the initial matching. The specific process of the OJ algorithm is shown in Algorithm 2. The *Top-K_A, Top-K_ J*, and *Top-K_ F* are the Top-K heaps of each partition, and *CM_A, CM_J*, and *CM_F* are the CM heaps of each partition. The starting edge and join order are determined in lines 3-4. As the candidate results, the results of *Top-K_A, Top-K_ J*, and *Top-K_ F* are stored in CES (line 5). According to the join order, the K results from CES are calculated and stored in Top-K_Result (line 7). The lower bound of each partition is calculated in line 10. Determine whether there are still the candidates to matching in lines 11–14. It continues to join the new matching results and compress with the initial Top-K results in lines 16–19. Repeat in this way until getting the final results.

Take Figure 9 as an example. First, traverse CES and determine the join order as Partition A ⟶ Partition J ⟶ Partition F. Then, the initial matching results of the query graph (v_1, v_2, v_3, v_4, v_5, v_6) are (u_{14}, u_7, u_6, u_8, u_1, u_9): 3.4 and (u_{14}, u_7, u_6, u_8, u_1, u_2):3.4, which get from joining the Top-K heaps of each partition, as shown in Figure 10. Then, calculate the lower bound values μ_i of each partition, respectively, $\mu_A = 3.4\text{-}1.2\text{-}1.0 = 1.2$, $\mu_J = 1.1$, and $\mu_F = 0.9$. Since the lower bound values are all not greater than the minimum interesting of the Top-K heap about the corresponding partition, the first K matching results are returned and made join again. Since there is no matching that the interestingness score is greater than 3.4 and each partition no longer generates a new result, the final matching results are (u_{14}, u_7, u_6, u_8, u_1, u_9):3.4 and (u_{14}, u_7, u_6, u_8, u_1, u_2):3.4. □

5. Experiments

In this section, we conduct extensive experiments to evaluate the proposed approach using real and synthetic datasets. We compare STF_ISQtop-k with PBSM [38], IEM [39], CECI [24], and several state-of-the-art

Input: *ECFIndex of query graph* E_Q, *CES, number of interesting subgraph K*
Output: *Top-K interesting subgraphs results R, the candidate matching heap CM*

```
(1)  CM = NULL, R = NULL;
(2)  int CP, O[|E_Q|];
(3)  O[0] ← First(CES);
(4)  O[ ] ← traverse E_Q from CES;
(5)  for each (traverse (u_i, u_j) in CES(O[0])) do
(6)     CP ← Size-c((u_i, u_j)) order by O[ ];
(7)     if (INT(Size-c())≤INT (R.bottom)) then
(8)        CM ← CP;
(9)        ascending (CM);
(10)    else then
(11)       CM ← R.bottom;
(12)       ascending (CM);
(13)       delete(R.bottom);
(14)       R ← CP;
(15)       descending (R);
(16) return R, CM;
```

Algorithm 1: UBV(*Size-c*()) matching.

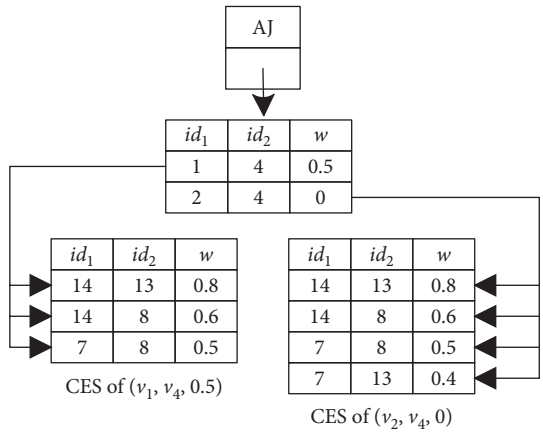

Figure 8: The candidate edge set (CES_J) of query subgraph Q_J.

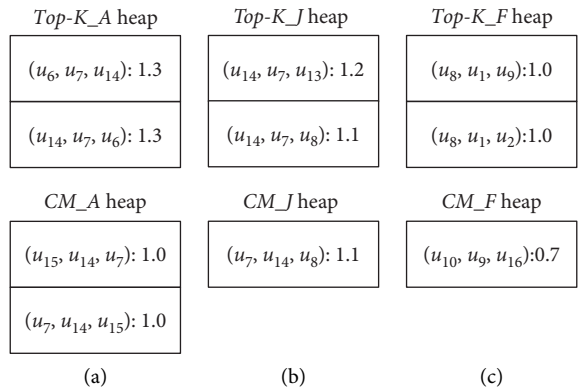

Figure 9: The Top-K heap and CM heap of each partition: (a) the heaps of subgraph (v_1, v_1, v_3) in partition A, (b) the heaps of subgraph (v_1, v_2, v_4) in partition J, and (c) the heaps of subgraph (v_4, v_5, v_6) in partition F.

Input: *Top-K_A, CM_A, Top-K_ J, CM_J, Top-K_ F, CM_F, Q*
Output: *Top-K_ Result*
(1) *Top-K_Result* = *NULL*, $i \in L_Q$;
(2) *int CP*, *O*[|*Partition*|], *CES*;
(3) *O*[*0*] ← *First*(*Q*);
(4) *O*[ ] ← *traverse Q*;
(5) *CES* ← *Top-K_A*; *Top-K_ J*; *Top-K_ F*;
(6) *clear*(*Top-K_A*, *Top-K_ J*, *Top-K_ F*);
(7) *Top-K_Result* ← *UBV(Size-c()) for CES order by O*[ ];
(8) μ = *Top-K_Result.bottom.INT*;
(9) **for each** $Q\_i \in Q$ **do**
(10) $\mu_i = \mu - \sum_{j=1, j \neq i}^{n} INT(r_{j,1})$;
(11) **if** $\exists(\mu_i < INT(Top\_i.top))$ **do**
(12) **for each** $\mu_i < INT(Top\_i.top)$ **do**
(13) **for each** *cm* in *CM_i* **do**
(14) *If INT*(*cm*) < μ_i
(15) *delete cm in CM_i*;
(16) *Top-K_i*, *CM_i* ← *continue UBV* (*Size-c*()) *in i partition and INT* > μ_i;
(17) **if** *Top-K_i NOT NULL* **do**
(18) *CES* ← *Top-K_i*;
(19) *Top-K.Result* ← *UBV(Size-c()) for CES order by O*[ ] *for new matching results from i partition*;
(20) **else then**
(21) **return** *Top-K_Result*;

Algorithm 2: Optimizational join.

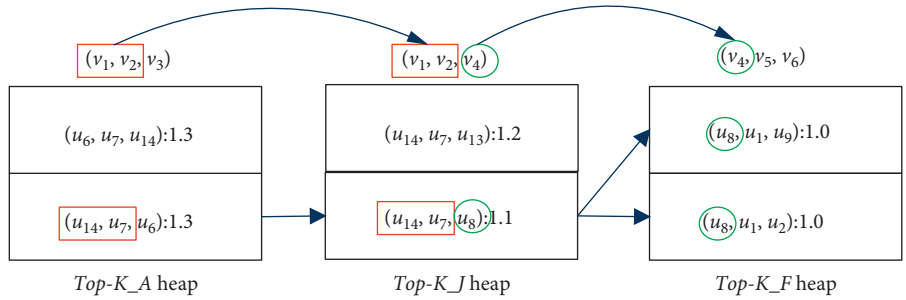

Figure 10: The initial matching results of query graph (v_1, v_2, v_3, v_4, v_5, v_6).

subgraph query solutions in labeled graphs. The literature [39] proposed two approaches IEM and CM for subgraph query. Since the performance of IEM was better than CM, in this paper, we choose IEM for comparison.

5.1. Experimental Settings. All experiments are performed on five Intel Pentium(R) CPU G3220@3.00GHZ machines with 8 GB of RAM and 1T hard disk drive. All experiments are implemented in JAVA. We repeat each experiment 5 times and report the average results.

5.1.1. Datasets. We use five different real datasets and multiple synthetic datasets to evaluate our approach. Real datasets include WordNet, DBLP, YouTube, DBpedia, and LiveJournal. WordNet, DBLP, and YouTube were used in [15, 36], DBpedia was used in [36], and LiveJournal was used in [27]. We obtained the DBLP, YouTube, and LiveJournal datasets from Stanford Large Network Dataset Collection (http://snap.stanford.edu/). WordNet represents relationships between English words, and labels of nodes represent different parts of speech. DBLP provides a comprehensive list of research papers in computer science. We chose the information of authors, fields, and journals to build the graph. DBpedia is crawled from Wikipedia. We cluster DBpedia into a data graph including the person dataset and their links, as in [36]. YouTube is extracted from a video-sharing website called YouTube. We chose the video IDs, uploaders, and categories to build the graph. LiveJournal is a free online blogging community where users declare friendship with each other, in which users have journals, individual blogs, and shared blogs. The details of the datasets above are given in Table 2. For DBpedia and LiveJournal, there are no given labels; as in [36], we have assigned a label for each vertex from a synthetic label set of sizes 50 and 100, respectively.

We generate four synthetic datasets using the R-MAT graph generator [42] with G_1, G_2, G_3, and G_4, and 10^4, 10^5, 10^6, and 10^7 are the number of vertices, respectively. The number of edges for each synthetic graph is $10|V|$. Labels of

Table 2: Statistics of real datasets.

Datasets	Abbr.	$\|V\|$	$\|E\|$	$\|\sum\|$
WordNet	WN	76854	213308	5
DBLP	DB	317080	1.04M	3
DBpedia	DP	809597	3.72M	50
YouTube	YT	1.13M	2.99M	3
LiveJournal	LJ	4.0M	34.68M	100

nodes are randomly assigned from A ~ E, and edge weights are randomly generated from [0, 1], which is referenced in [13].

5.1.2. Query Sets. We ensure that each query has at least one matching in the data graph. We traverse the data graph by depth-first search (DFS) to generate the connected query graphs of size from 2 to 50 nodes similar to existing works [15, 24, 38, 39]. Then, we assign labels and add query constraints randomly. Unless specified explicitly, we are interested in computing the top 10 interesting subgraphs ($K = 10$).

5.2. Experimental Analysis

5.2.1. Preprocessing Time and Memory Cost. Preprocessing is crucial to the subgraph query as it can significantly reduce the memory consumption, as well as the total runtime. Figures 11(a) and 11(b) show the comparison of preprocessing time on different sizes of real datasets and synthetic datasets.

The necessary preprocessing tasks for CECI include finding the root query node, generating the query tree, determining the matching order, breaking automorphism, and finding the cluster pivots. Since preprocessing is mainly aimed at the query graph, we choose a compromised query graph size of 25 nodes. CECI requires less processing time than performing the data graph due to the smaller scale.

The preprocessing of PBSM includes data graph compression, in which two vertices are equivalent. It traverses the data graph to find the equivalent vertices and merges them to reduce the size of the data graph. Since it does not involve a large amount of computation, it has the shortest preprocessing time compared with other methods.

The method in [39] includes the creation of the DVI index, which includes destination vertex count (DVC) and sorted edge lists. To obtain all DVC, it uses breadth first search (BFS) from one vertex to another at distance D. In supplement to DVC, it keeps sorted edge lists. The preprocessing time of DVI is affected by D.

The preprocessing time of the method proposed in this paper includes data graph compression, STFIndex creation and STFIndex partition. Thus, compared with PBSM and DVI, it needs more preprocessing time. Since the preprocessing operations in the last three methods are independent of the query graph, only the data graph is preprocessed. Thus, the operations can be completed offline without affecting the efficiency of the online query.

Figure 12 shows the comparison of memory cost among STIndex, PBSM, DVI, and CECI on different datasets. In this paper, STIndex will replace the storage of the original data graph, and the memory cost only STIndex without extra storage. The memory cost of PBSM includes the information of the data graph and the compressed graph. The memory cost of it is affected by the compression ratio. DVI includes the data graph and the information of the DVI index, which are distention vertex count and sorted edge lists. According to the increase of D, more information of neighbors needs to be stored, and the memory cost of DVI will increase exponentially with the increase of D. CECI needs to store data graphs, the information of preprocessed query graph as well as the candidates of the query graph. We choose the query graph with 5 nodes in our experiments. It still needs more memory space.

5.2.2. Parallel Processing Efficiency. In this section, we evaluate the efficiency of the STFIndex partition. We compare the parallel processing and single machine solution with the increasing number of query graph size on two real datasets DBLP and LiveJournal, and a synthetic dataset G_3.

As shown in Figure 13, the abscissa is the number of nodes in the query graph, and the ordinate is the acceleration ratio between the query efficiency after partition and the single-machine query. We can see the larger the data graph and the more labels in the data graph, the more obvious the speedup is.

The main reason for the speedup is that the single-machine processing solution needs to be verified one by one in the complete index according to the matching principle to obtain candidate nodes and edge sets. However, when the index is partitioned, we can search the candidate nodes and edges from different partitions in parallel. Since the partition method in this paper is partitioned according to the node type, the relevant edge information can be copied to the partition at the same time. This can reduce the communication overhead between different partitions effectively, and the parallel efficiency is relatively high. The trend slightly of the speedup ratio flats out sometimes. This is because, with the increasing of the query graph, parallel processing needs more time to join the subresults to obtain the final subgraphs.

5.2.3. Running Time of Personalized Top-K Interesting Subgraph Query. We have run the experiment to find the top ten ($K = 10$) subgraphs of the query graph without weight (Q) as in Figure 1(b), and the query graph with weight (Q') as in Figure 7 on five different real datasets and large synthetic dataset G_4. Figure 14 shows the speedup ratio of running time for the proposed method and IEM, PBSM, and CECI.

As shown in Figure 14, STF_ISQtop-k outperforms IEM by an average of 1.75 and 5.02 on Q and Q', respectively. The main reason for the speedup is that IEM utilizes the DVI index to filter candidates one by one, which does not consider the potential value of isotype nodes. The speedup on Q' is larger than on Q. This is because IEM needs to

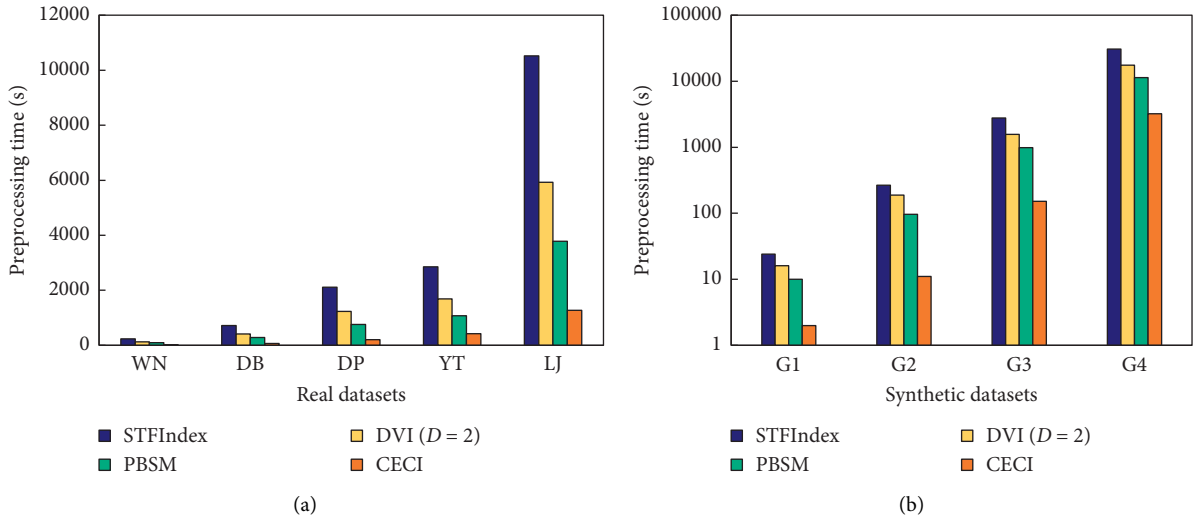

Figure 11: Comparison of preprocessing time on real datasets (a) and synthetic datasets (b).

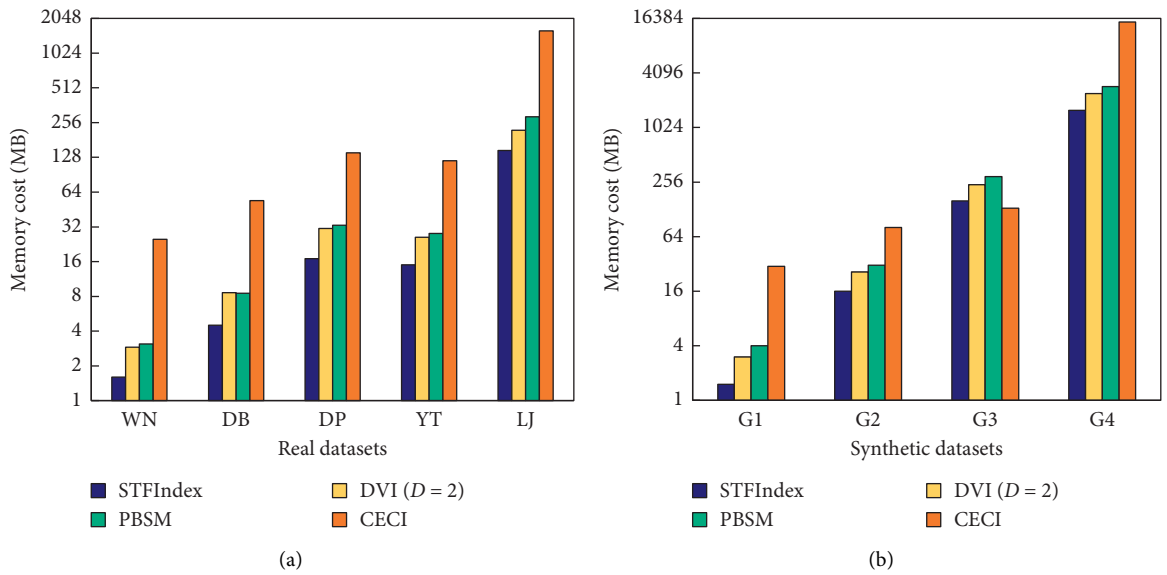

Figure 12: Comparison of memory cost on real datasets (a) and synthetic datasets (b).

search all isomorphism subgraphs in the data graph and rank the results to choose the top ten, which takes more time. However, STF_ISQtop-k has pruned edges whose weights do not meet the constraints in the filtering stage, which avoids repeated calculation. It is more suitable for weighted graph query. Both PBSM and STF_ISQtop-k take into account the problem of isotype nodes and introduce graph compression, which can reduce the size of the data graph and filter the invalid candidates effectively. However, STF_ISQtop-k implements parallel calculating based on the partition STFIndex in the phase of filter, rather than traverse the entire supergraph as in PBSM.

In addition, since PBSM needs to filter subgraphs whose existing weights do not meet the query constraints from all query results, the filtering is repeated. Thus, STF_ISQtop-k is better than PBSM for both Q and Q', especially on the weighted query graph Q'. The running time of CECI includes the time for preprocessing, CECI creation, and the enumeration of the subgraphs. Although the preprocessing of CECI is performed online, the parallel technique speeds up its query.

5.2.4. Impact of Query Graph Size and K. We evaluate the impact of query graph size on the running time of interesting subgraph query. We restrict the comparison with two real datasets WordNet and YouTube and assign weights to the query graphs randomly. The running time of each method increases with the growth of the query graph size.

As shown in Figure 15, STF_ISQtop-k has the most obvious advantages over IEM, and this is especially true in the large graph. This is because IEM needs to verify the result set again and filter the subgraphs that the edge weights

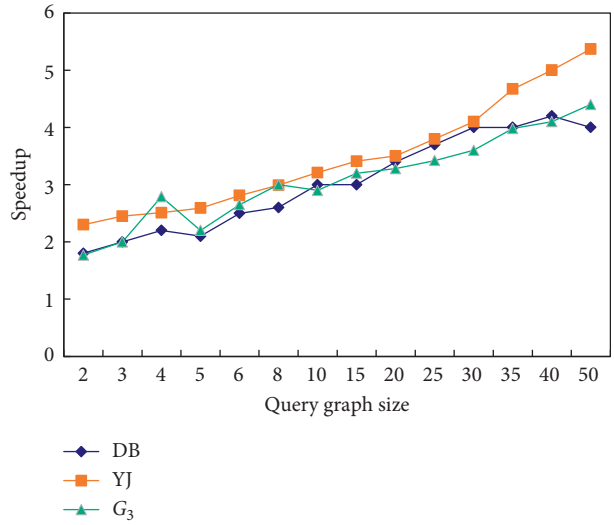

Figure 13: Speedup obtained by parallel processing on different datasets.

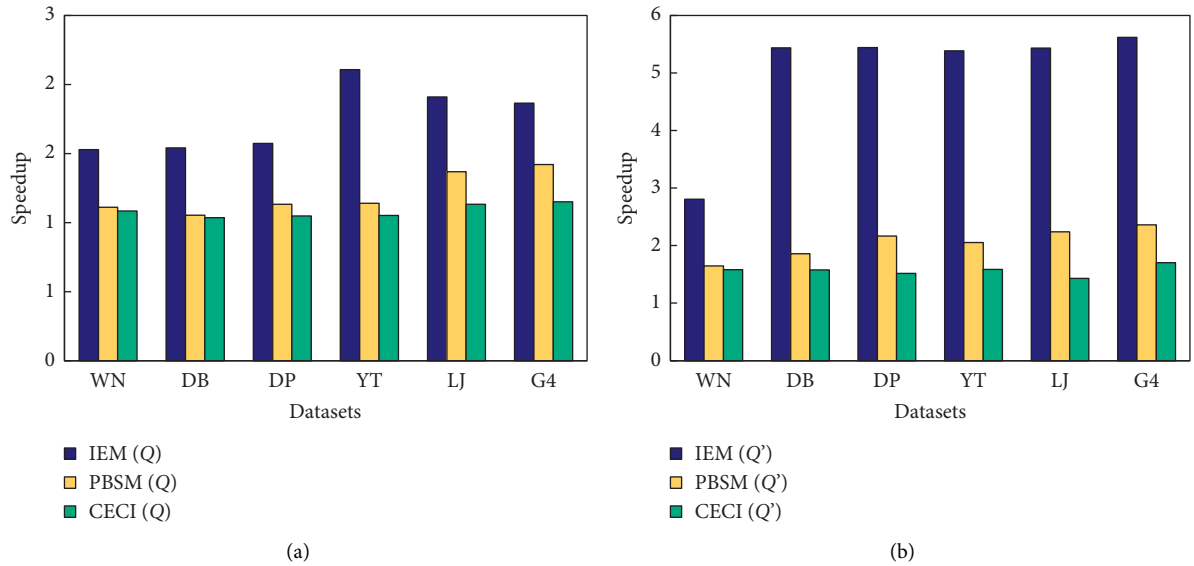

Figure 14: Performance comparison of interesting subgraph query time on the query graph without weight Q (a) and with weight Q' (b).

satisfy the query conditions. Meanwhile, as the size of the query graph increases, the number of iterations increased, which leads to the greater query time increases significantly. Compared with IEM, PBSM has better query performance utilizing compression technology. It also lacks the optimal query strategy for the weighted query graph, which only filters the results again to obtain the satisfying final subgraphs. CECI adopts parallel processing technology to effectively improve the query time, but it requires creating a CECI index for each query graph. However, the method in this paper only creates the index of the data graph offline, which can be used many times after the construction is completed. CECI does not consider homotype node-compression, which requires filtering for each query node to extract candidate nodes. The speedup of our STF_ISQtop-k comes from the reduction of redundancy in filtering, which prunes the unsatisfied subgraphs. The larger the size of the query graph, the more nodes and edges are compressed out, and the advantages of STF_ISQtop-k are more obvious.

Figure 16 shows the impact of changed K for our STF_ISQtop-k, which is verified on real dataset DBLP. It can be seen from the figure that the query time increases with the

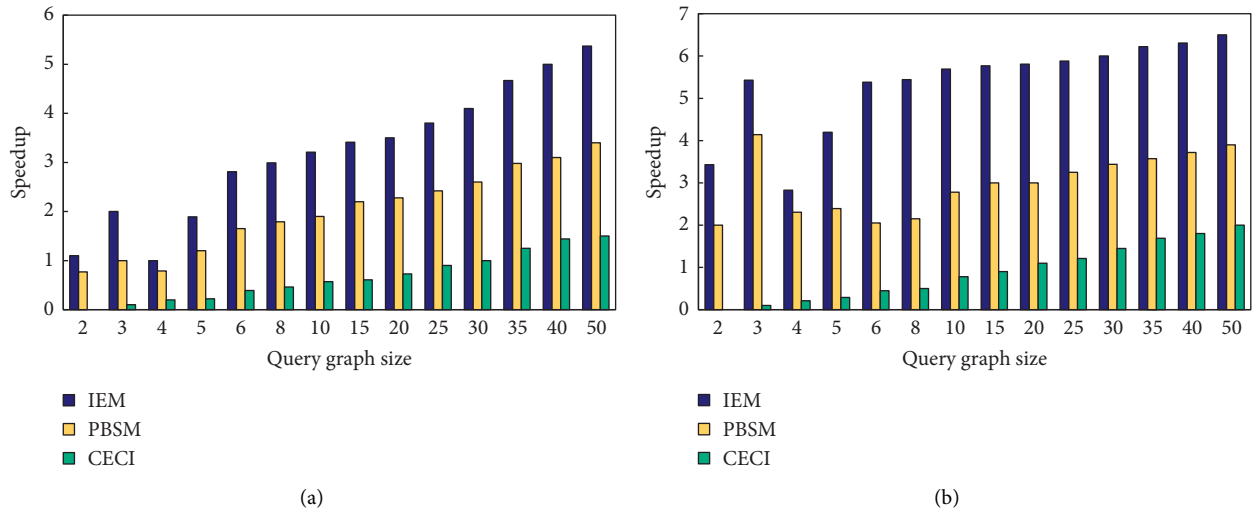

Figure 15: Performance comparison of interesting subgraph query time on WordNet (a) and YouTube (b) with different query graph size.

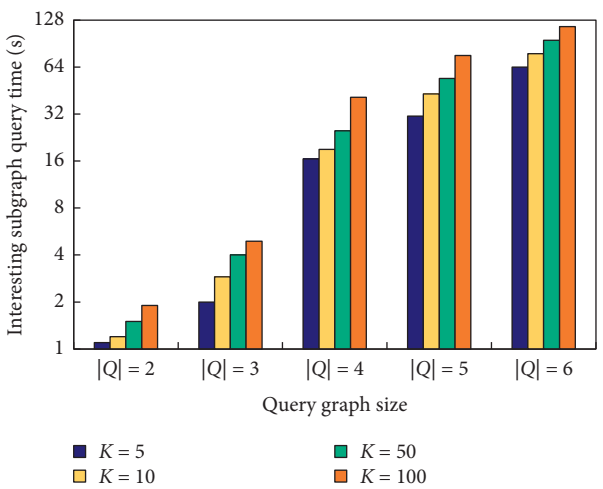

Figure 16: Interesting subgraph query time for different values of K in DBLP.

increase of query graph size, but the increase is relatively gradual. And when the query graph Q is fixed and K changes, the fluctuation of the query time is small. So, it is verified that the STF_ISQtop-k has good scalability.

6. Conclusions

In this paper, we propose a novel intelligent solution for the Top-K interesting subgraph query. Specifically, the proposed ISGC strategy can compress the data graph to a smaller size and filter multiple invalid nodes and edges in batch. The proposed auxiliary data structure STFIndex represents all topology information in the compressed graph, which can replace the storage of the original data graph. Meanwhile, STFIndex will be created offline to improve the efficiency of an online query. We also propose a partition method based on ELSV to divide the STFIndex into several partitions, which realizes the parallel query in each partition. With the help of STFIndex, we provide a personalized Top-K interesting subgraph query approach in large labeled graphs named STF_ISQtop-k. It contains MDF strategy, UBV (Size-c) matching, and the OJ method, which are used to filter out as many false candidates as possible to achieve fast joins of subquery results. The experimental results show that the proposed approach can perform Top-K interesting subgraph query in large labeled graphs efficiently, and the query results have practical significance in practical applications. In the future, we will apply our solution to solve the problems in other areas and improve our Top-K interesting subgraph query approach.

Data Availability

The data used to support the findings of this study are available from visiting http://snap.stanford.edu/.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by the National Key Research and Development Program of China (grant no. 2019YFC0850103), National Natural Science Foundation of China (grant nos. 61472169, 61502215, 62072220, and U1811261), China Postdoctoral Science Foundation Funded Project (grant no. 2020M672134), Science Research Fund of Liaoning Province Education Department (grant no. LJC201913), and Liaoning Public Opinion and Network Security Big Data System Engineering Laboratory (grant nos. 04-2016-0089013).

References

[1] A. B. Sonmez and T. Can, "Comparison of tissue/disease specific integrated networks using directed graphlet signatures," *BMC Bioinformatics*, vol. 18, no. 4, pp. 41–50, 2017.

[2] L. Bingxian, Z. Zhicheng, W. Lei et al., "Confining Wi-Fi coverage: a crowdsourced method using physical layer information," in *Proceedings of the 13th Annual IEEE*

International Conference on Sensing, Communication, and Networking (SECON), pp. 1–9, London, UK, June 2016.

[3] B. Lu, L. Wang, J. Liu et al., "LaSa: location aware wireless security access control for IoT systems," *Mobile Networks and Applications*, vol. 24, no. 3, pp. 748–760, 2019.

[4] X. J. Li and G. H. Yang, "Graph theory-based pinning synchronization of stochastic complex dynamical networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 2, pp. 1–11, 2017.

[5] Tencent Technology, "WeChat's monthly active users had 1.12 billion, increased year-one-year by 6.9%," 2020, https://tech.qq.com/a/20190515/007600.htm.

[6] Tencent Technology, "Facebook's monthly active users had 2.45 billion, income increased year-one-year by 30% in the third quarter," 2020, https://tech.qq.com/a/20191031/001015.htm.

[7] E. Abdelhamid, I. Abdelaziz, Z. Khayyat, and P. Kalnis, "Pivoted subgraph isomorphism: the optimist, the pessimist and the realist," in *Proceedings of the 22nd International Conference on Extending Database Technology, EDBT 2019*, pp. 361–372, Lisbon, Portugal, March 2019.

[8] X. Wang, Q. Xu, L. L. Chai, Y. J. Yang, and Y. P. Chai, "Efficient distributed query processing on large scale RDF graph data," *Journal of Software*, vol. 30, no. 3, pp. 498–514, 2019.

[9] B. Wu and H. Shen, "Exploiting efficient densest subgraph discovering methods for big data," *IEEE Transactions on Big Data*, vol. 3, no. 3, pp. 334–348, 2017.

[10] M. Abulaish, Z. A. Ansari, and S. Jahiruddin, "SubISO: a scalable and novel approach for subgraph isomorphism search in large graph," in *Proceedings of the 11th International Conference on Communication Systems and Networks (COMSNETS)*, pp. 1–8, Bangalore, India, January 2019.

[11] L. Lai, Z. Qing, Z. Yang et al., "Distributed subgraph matching on timely dataflow," *Proceedings of the VLDB Endowment*, vol. 12, no. 10, pp. 1099–1112, 2019.

[12] J. Kim, H. Shin, W.-S. Han, S. Hong, and H. Chafi, "Taming subgraph isomorphism for RDF query processing," in *Proceedings of the VLDB Endowment*, pp. 1238–1249, Kohala Coast, HI, USA, February 2015.

[13] M. Gupta, J. Gao, X. Yan, H. Cam, and J. Han, "Top-K interesting subgraph discovery in information networks," in *Proceedings of the IEEE 30th International Conference on Data Engineering, ICDE*, pp. 820–831, Chicago, IL, USA, March 2014.

[14] X. Shan, C. Jia, L. Ding, X. Ding, and B. Song, "Dynamic top-K interesting subgraph query on large-scale labeled graphs," *Information*, vol. 10, no. 2, pp. 61–83, 2019.

[15] J. R. Ullmann, "An algorithm for subgraph isomorphism," *Journal of the ACM*, vol. 23, no. 1, pp. 31–42, 1976.

[16] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, New York, NY, USA, 1979.

[17] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub) graph isomorphism algorithm for matching large graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 10, pp. 1367–1372, 2004.

[18] L. B. Holder, D. Cook, and S. Djoko, "Substructure discovery in the SUBDUE system," in *Proceedings of the Workshop on Knowledge Discovery in Databases, KDD'94*, pp. 169–180, Seattle, WA, USA, July 1994.

[19] F. Zhu, Q. Qu, D. Lo, X. Yan, J. Han, and P. S. Yu, "Mining top-K large structural patterns in a massive network," in *Proceedings of the 37th International Conference on Very Large Data Bases, VLDB'11*, pp. 807–818, Seattle, WA, USA, August 2011.

[20] A. Ray, L. B. Holder, and A. Bifet, "Efficient frequent subgraph mining on large streaming graphs," *Intelligent Data Analysis*, vol. 23, no. 1, pp. 103–132, 2019.

[21] H. He and A. K. Singh, "Query language and access methods for graph databases," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data, SIGMOD 2008*, pp. 405–418, Vancouver, Canada, June 2008.

[22] P. Zhao and J. Han, "On graph query optimization in large networks," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 340–351, 2010.

[23] F. Bi, L. Chang, X. Lin et al., "Efficient subgraph matching by postponing Cartesian products," in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD '16*, pp. 1199–1214, San Francisco, CA, USA, June 2016.

[24] B. Bhattarai, H. Liu, and H. Howie Huang, "CECI: compact embedding cluster index for scalable subgraph matching," in *Proceedings of the 2019 International Conference on Management of Data, SIGMOD'19*, pp. 1447–1462, Amsterdam, Netherlands, June 2019.

[25] H. He and A. K. Singh, "Filtering strategies for inexact subgraph matching on noisy multiplex networks," in *Proceedings of the 2019 IEEE International Conference on Big Data*, pp. 4906–4912, Los Angeles, CA, USA, December 2019.

[26] M. Han, H. Kim, G. Gu et al., "Efficient subgraph matching: harmonizing dynamic programming, adaptive matching order, and failing set together," in *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data, SIGMOD '19*, pp. 1429–1446, Amsterdam, Netherlands, June 2019.

[27] Y. Park, S. Ko, S. S. Bhowmick et al., "G-CARE: a framework for performance benchmarking of cardinality estimation techniques for subgraph matching," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD '20*, pp. 1099–1114, Portland, OR, USA, June 2020.

[28] N.-T. Le, B. Vo, L. B. Q. Nguyen, H. Fujita, and B. Le, "Mining weighted subgraphs in a single large graph," *Information Sciences*, vol. 514, pp. 149–165, 2020.

[29] W. S. Han, J. Lee, and J. H. Lee, "TurboISO: towards ultrafast and robust subgraph isomorphism search in large graph databases," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD'13*, pp. 337–348, New York, NY, USA, June 2013.

[30] X. Ren and J. Wang, "Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs," *Proceedings of the VLDB Endowment*, vol. 8, no. 5, pp. 617–628, 2015.

[31] H. Zhang, X. Xie, Y. Duan, and Y. Wen, "An algorithm for subgraph matching based on adaptive structural summary of labeled directed graph data," *Chinese Journal of Computers*, vol. 40, no. 1, pp. 52–71, 2017.

[32] C. Nabti and H. Seba, "Querying massive graph data: a compress and search approach," *Future Generation Computer Systems*, vol. 74, pp. 63–75, 2017.

[33] X. Yan, B. He, F. Zhu, and J. Han, "Top-K aggregation queries over large networks," in *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010*, pp. 377–380, Long Beach, CA, USA, March 2010.

[34] S. Yang, F. Han, Y. Wu, and X. Yan, "Fast top-K search in knowledge graphs," in *Proceedings of the 2016 IEEE 32nd International Conference on Data Engineering, ICDE'16*, Helsinki, Finland, May 2016.

[35] Z. Sun, H. Huo, and X. Chen, "Fast top-K graph similarity search via representative matrices," *IEEE Access*, vol. 99, p. 1, 2018.

[36] Z. Yang, A. W. Fu, and R. Liu, "Diversified top-K subgraph querying in a large graph," in *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data, SIGMOD'16*, pp. 1167–1182, San Francisco, CA, USA, June 2016.

[37] P. Fournier-Viger, C. Cheng, J. C.-W. Lin, U. Yun, and R. U. Kiran, "TKG: efficient mining of top-K frequent subgraphs," in *Proceedings of the 7th International Conference on Big Data Analytics, BDA 2019*, pp. 209–226, Ahmedabad, India, December 2019.

[38] W. Chen, J. Liu, Z. Chen, X. Tang, and K. Li, "PBSM: an efficient top-K subgraph matching algorithm," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 32, no. 6, 2018.

[39] N. Amin, K. U. Khan, B. Dolgorsuren, and Y. K. Lee, "Extracting top-K interesting subgraphs with weighted query semantics," in *Proceedings of the IEEE International Conference on Big Data and Smart Computing, BigComp*, pp. 366–373, Jeju Island, South Korea, February 2017.

[40] A. Bendimerad, A. Mel, J. Lijffijt, M. Plantevit, C. Robardet, and T. D. Bie, "Mining subjectively interesting attributed subgraphs," in *Proceedings of the 14th International Workshop on Mining and Learning with Graphs (MLG)*, London, UK, May 2018.

[41] N. Jayalakshmi, P. Padmaja, and G. J. Suma, "An approach for interesting subgraph mining from web log data using W-Gaston algorithm," *International Journal of Uncertainty, Fuzziness, and Knowledge-Based Systems*, vol. 27, no. 2, pp. 277–301, 2019.

[42] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: a recursive model for graph mining," in *Proceedings of the Siam International Conference on Data Mining, SDM'04*, pp. 442–446, Lake Buena Vista, FL, USA, April 2004.