World Wide Web (2016) 19:417–448
DOI 10.1007/s11280-014-0322-0

Answering subgraph queries over massive disk resident graphs

Peng Peng · Lei Zou · Lei Chen · Xuemin Lin · Dongyan Zhao

Received: 4 February 2013 / Revised: 8 December 2014 /
Accepted: 22 December 2014 / Published online: 25 January 2015

Abstract Due to its wide applications, *subgraph query* has attracted lots of attentions in database community. In this paper, we focus on subgraph query over a single large graph G, i.e., finding all embeddings of query Q in G. Different from existing feature-based approaches, we map all edges into a two-dimensional space $\Re^2$ and propose a bitmap structure to index $\Re^2$. At run time, we find a set of *a*djacent *e*dge *p*airs (AEP) or *s*tar-*s*tyle *p*atterns (SSP) to cover Q. We develop *e*dge *j*oin (EJ) algorithms to address both AEP and SSP subqueries. Based on the bitmap index, our method can optimize I/O and CPU cost. More importantly, our index has the linear space complexity instead of exponential complexity in feature-based approaches, which indicates that our index can scale well with respect to large data size. Furthermore, our index has light maintenance overhead, which has not been considered in most of existing work. Extensive experiments show that our method significantly outperforms existing ones in both online and offline processing with respect to query response time, index building time, index size and index maintenance overhead.

Keywords Subgraph match · Bitmap index · Edge join

P. Peng · L. Zou (✉) · D. Zhao
Peking University, Beijing, China
e-mail: zoulei@pku.edu.cn

P. Peng
e-mail: pku09pp@pku.edu.cn

D. Zhao
e-mail: zhaody@pku.edu.cn

L. Chen
Hong Kong University of Science and Technology, Hong Kong, China

X. Lin
University of New South Wales, New South Wales, Australia

1 Introduction

Due to the flexibility of graphs, more and more applications adopt graphs as the underlying model, such as communication networks, biological networks and social networks. Hence, graph databases have recently gained lots of attentions in database community [4, 13, 16, 19, 24, 26, 27, 32, 33, 36, 39, 40, 43]. Graph databases represent and store information by nodes and connecting edges and the key feature in graph databases is that query processing is optimized for structural queries, such as shortest-path queries [2, 5, 6, 17], reachability queries [3, 6, 29, 31], and subgraph queries [24, 33]. In this paper, we focus on *subgraph queries*. Generally speaking, there are two scenarios of graph database models in these literatures. The first scenario is that graph database has a large number of small-size connected data graphs, i.e., the graph-transaction database. Given a query Q, subgraph query retrieves all data graphs containing Q. For example, scientists want to find all molecules having a specified substructure (such as benzene ring) from a compound database. In the second scenario, there is a single large graph (may not be connected) G, such as biological networks and social networks. Given a query Q, subgraph query needs to locate all embeddings of Q in G. For example, given a biological network G and a structural motif Q, we want to locate all embeddings of Q. Usually, in the second scenario, the size of the largest connected component in G is very large.

In this paper, we focus on the second scenario, i.e., finding all embeddings of Q over a single large graph G. The hardness of this problem lives in its exponential search space. Obviously, it is impossible to employ some subgraph isomorphism algorithm, such as ULLMANN [30] and VF2 [7], over a very large graph to find all embeddings on the fly. In order to speed up query processing, we need to create indices for large graphs and rely these indices to reduce the search space. Apparently, these indices should be small and have light maintenance cost.

A possible solution is to find some frequent substructures (such as paths, subtrees and subgraphs) as *features*. Then, for each feature, we maintain a list of its embedding positions. The feature-based approach is often used in graph-transaction database. Unfortunately, this approach cannot work well in a single large graph G. As far as we know, mining frequent subgraphs in a large graph is still an open question in data mining community.

Furthermore, pre-computed indices should have the light maintenance overhead. Most exiting methods assume that graph databases are static or updated in batch. However, the assumption cannot hold in some applications. For example, the individual relationships are always changing in social networks. In this case, the desirable indexing structures should have light maintenance overhead. For feature-based index approaches, when the structural information of a vertex in a single large graph is updated, since the update operation may involve lots of vertices and features, it is quite expensive to update the embedding lists of all involved features. For example, GADDI [37] tries to find some discriminative discriminating substructures and uses these substructures to define a novel distance. If there are some updates in the data graph, because the frequencies of some discriminating substructures are changed, the distances need to recalculated.

In this paper, we propose an index that can meet the above two requirements–small index size and light maintenance cost. Firstly, we map all edges into two-dimensional points, namely, converting a data graph into a two-dimensional space $\Re^2$. Then, according to edge endpoint labels, we re-arrange $\Re^2$ and partition it into different areas. All points in $\Re^2$ corresponding to edges with the same endpoint labels will be in the same area. For each area, we assign two bitstrings to summarize all edges in this cluster. Given a large graph G,

the space complexity of the indexing structure is $O(|E(G)|)$, where $|E(G)|$ is the number of edges in G. Furthermore, our index maintenance algorithm has linear time complexity with respect to number of insertions or deletions.

At run time, given a query Q, we first find a set of adjacent edge pairs (AEP) or star-style pattern (SSP) subqueries to cover Q. Then, we develop two kinds of edge join algorithms to address AEP and SSP subqueries, respectively. Finally, we perform a series of two-way joins to find final matches for Q. Here, for finding good subqueries to cover Q, we will introduce a cost model.

To summarize, in this work, we have made the following contributions:

1) We propose a novel bitmap index for a large graph G, which has both linear space cost and light maintenance overhead.
2) We find a set of AEP subqueries to cover a query Q. Based on our proposed bitmap index, we develop an efficient *E*dge *J*oin (EJ) algorithm to answer AEP subqueries. We propose a cost model and a histogram-based cost estimation method to guide AEP selection.
3) In order to further improve query performance, we propose to use SSP instead of AEP subqueries. We also develop a bitmap-based method to reduce the intermediate result size.
4) Finally, we conduct extensive experiments over both real and synthetic datasets to evaluate our proposed approaches.

The rest of the paper is organized as follows. Related work is discussed in Section 2. The problem definition is given in Section 3. A bitmap index is proposed in Section 4. Section 5 proposes an AEP-based query algorithm. In order to improve the query performance, in Section 6, we propose two optimization techniques: SSP-based solution and reducing intermediate result size by bitmap indices. The index maintenance method is discussed in Section 7. Section 8 shows how to extend our method for handling the undirected graphs. We report the effectiveness of proposed methods through extensive experiments in Section 9 and conclude the paper in Section 10.

2 Related work

Subgraph search is a fundamental operation in graph databases. So far, there have been lots of proposals for subgraph search problem [4, 13, 16, 24, 26, 27, 32, 33, 36, 39, 40, 43]. As mentioned earlier, there are two scenarios of graph databases, thus, in this section, we will survey some related work in these two scenarios, respectively.

In the first scenario, i.e., a graph database having a large number of small-size graphs, most existing methods adopt "filter-and-refine" framework. Specifically, first some pruning rules are adopted to filter out a large number of false positives. Then, the final results are fixed by subgraph isomorphism checking over candidates. In this way, the online performance depends on the pruning power of different pruning strategies. The most popular pruning method is "feature-based" pruning, which uses some paths, subtrees or subgraphs as structural features [4, 19, 24, 33, 36]. Assume that query Q has a structural feature F. All data graphs without containing F will be pruned safely. For example, all paths up to length $maxL$ ($maxL$ is a specified parameter) are selected as features in GraphGrep [24]. In order to improve pruning power, gIndex [33] and FGIndex [4] employ graph mining techniques to find some frequent subgraphs as features. Recently, CT-index [19] have proposed a hash-

key fingerprint technique based on trees and cycles features to filter candidates. However, it is quite difficult to extend feature-based approaches into a single large graph (i.e., the second scenario) due to two reasons. First, finding frequent subgraphs in a single large graph is still an open problem in data mining community. Second, the space cost of feature-based approaches is expensive, since all feature embedding positions need to be recorded.

Different from feature-based methods, there are some other approaches that do not employ structural features as indexing elements, such as Closure-tree [13] and GCode [44]. In Closure-tree, authors propose pseudo subgraph isomorphism by checking the existence of a semi-perfect matching from vertices in query graph to vertices a data graph (or graph closure). However, when the data graph size is very large, finding semi-perfect matching is expensive. Thus, this method cannot work well in the second scenario of graph databases. In our earlier work GCode [44], we compute vertex signature based on the local structure of the graph [44]. The filtering strategy is based on the interlacing theorem in spectral graph theory.

Recently, subgraph search over a single large graph has began to attract researchers' attentions, such as GADDI [37], Nova [42] and SPath [39]. GADDI, Nova [42], SPath [39] try to construct some indices to prune the search space of each vertex, such that the whole search space can be jointly reduced as much as possible. GADDI proposes an index based on *n*eighboring *d*iscriminating *s*ubstructure (NDS) distances, which need to count the number of some small discriminating substructures in the intersecting subgraph of each two vertices. Nova proposes an index named *nIndex*, which is based on the label distribution and is integrated into a vector domination model. Both GADDI and Nova are memory-based algorithm, meaning that they cannot scale to very large graphs. For example, GADDI cannot work when $|V(G)| > 40K$, as shown in Figure 13, and Nova cannot work when vertex degree is larger than 10, as shown in Figure 14. SPath constructs an index by neighborhood signature, which utilizes the shortest paths within the k-neighborhood subgraph of each vertex in the data graph. Because the index is built based on the k-neighborhood subgraph of each vertex, the index building cost is very high, especially for a large graph. Furthermore, SPath does not address the update issues. Distance-join [43] is our earlier work, in which, we propose a distance join algorithm for pattern match query over a single large graph. The match definition in [43] is not subgraph isomorphism as defined in Definition 2 in this paper. Thus, the method in [43] cannot be used to answer subgraph query problem.

Some methods study the semantics of the subgraph query in a more general model, such as GraphQL [14] and G-SPARQL [22].They propose query languages for graph databases that support arbitrary attributes on nodes, edges and graphs. However, none of them discusses how to construct structural indices in the general graphs to speed up query processing. GraphQL suggest some optimization strategies, but does not discuss them in detail.

In [20], the authors made a thorough comparison of existing solutions, such as GraphQL, GADDI, SPath and so on. The experiments show that there was no single winner for all experiments, but SPath is better than others in some cases. To overcome the shortcomings of existing methods, [12] proposes a new approach named TurboISO. TurboISO firstly rewrite the query and divide all query vertices into some *Neighborhood Equivalence Classes*. Two query vertices in the same neighborhood equivalence classes can map to the same vertex. Then, TurboISO does BFS search over the rewritten query from a selected query vertex u and find a tree T_g. Then, TurboISO does DFS search from u's candidate to determine a candidate region. The depth of DFS search is based on T_g. Finally, TurboISO finds final matches over the candidate region based on the matching order. TurboISO employs DFS

search from some vertices. If the data graph is dense which means the diameter of the data graph is small, the candidate region of a vertex may be very large. Then, TurboISO still takes much time to join many candidates of the large candidate region.

Furthermore, Sun et al. [25] use Microsoft's distributed graph database system, Trinity [23], to answer the subgraph query, while Gao et al. [11] use an open source distributed graph database system, Giraph [15], to approximately answer the subgraph query. Unlike most of methods based on multi-way join of candidates, these method utilize efficient graph exploration for query processing.

Sometimes, the notion of exact subgraph search are too restrictive in some applications, so many works [8–10, 18, 28, 34, 35, 38, 41] revise the notion. Fan et al. [8–10] use graph simulation to catch sensible matches that traditional notions of subgraph search fail to identify. Ness [18] proposes a graph similarity measure in an information propagation model. Then, Ness do subgraph similarity search under this measure. G-Ray [28] defines the subgraph similarity search based on the model of random walk with restart. In [35, 38, 41], researchers utilize the minimum graph edit distance constraint to define the graph similarity search problem. Given a query graph Q, they try to find all subgraphs whose edit distances to Q are smaller than a threshold. In [34], the authors find matches that map vertices and edges of a query via some transformation functions in a graph.

3 Background

In this section, we review the terminology that we will use in this paper, and formally define our problem. In this work, we study subgraph search over a large *directed vertex-labeled* graph (Definition 1). In the following, unless otherwise specified, the term "graph" refers to a directed vertex-labeled graph. Note that, we will discuss how to extend our method to handle a large "undirected" graph in Section 8.

Definition 1 A *directed vertex-labeled graph* G is denoted as $G = (V(G), E(G), L, F)$, where (1) $V(G)$ is a set of vertices, and (2) $E(G) \subseteq V \times V$ is a set of directed edges, and (3) L is a set of vertex labels, and (4) the labeling function F defines the mapping $F : V(G) \rightarrow L$.

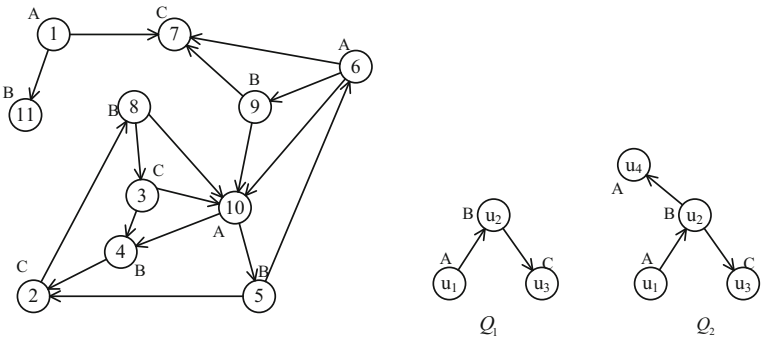
Furthermore, according to the alphabetical order, we can define the *total order* for all distinct vertex labels in L.

Figure 1a shows a running example of a directed vertex-labeled graph. Note that, the numbers inside the vertices are *vertex IDs* that we introduce to simplify description of the graph; and the letters beside the vertices are *vertex labels*. A directed edge from v_1 to v_2 is denoted as $\overrightarrow{v_1 v_2}$.

Definition 2 A labeled graph $G = (V(G), E(G), L, F)$ is *isomorphic* to another graph $G' = (V'(G'), E'(G'), L', F')$, denoted by $G \approx G'$, if and only if there exists a bijection function $g : V(G) \rightarrow V'(G')$ *s.t.*

$$1) \forall v \in V(G), F(v) = F'(g(v)); \textit{and}$$
$$2) \forall v_1, v_2 \in V(G), \overrightarrow{v_1 v_2} \in E \Leftrightarrow \overrightarrow{g(v_1) g(v_2)} \in E'$$

Given two graphs Q and G, Q is *subgraph isomorphic* to G, denoted as $Q \subseteq G$, if Q is *isomorphic* to at least one subgraph G' of G, and G' is a *match* of Q in G.



(a) Data Graph G (b) Query Graph Q_1 and Q_2

Figure 1 Graph G and query Q

Definition 3 *(Problem Statement)* Given a large data graph G and a query graph Q, where $|V(Q)| \ll |V(G)|$, the problem that we conduct in this paper is defined as to find all *matches* of Q in G, where matches are defined in Definition 2.

Table 1 shows some frequently-used notations in this paper.

4 Index

As mentioned in Section 1, the main idea of our method is that we classify all edges into different groups according their endpoint labels. Then, for each group, we assign two bit-strings (called signatures) to summary all edges in this group, and store the edges according to the order of starting and ending vertex, respectively. Given a query graph, we can first

Table 1 Notations

$G = (V(G), E(G), L, F)$	Data graph
$Q = (V(Q), E(Q), L', F')$	Query graph
v, u	Vertex in data graph and query graph, respectively
l	Label
$< l_1, l_2 >$	A label pair
$F^{-1}(l)$	All vertices in G with the label l
$F^{-1}(\langle l_1, l_2\rangle)$	All edges in G with the label pair $\langle l_1, l_2\rangle$
$F^{-1}(\langle l_1, l_2\rangle)\|_{l_1}$	All vertices with the label pair l_1 and adjacent to an edge with the label pair $\langle l_1, l_2\rangle$
$SB(\langle l_1, l_2\rangle), EB(< l_1, l_2 >)$	Start signature and end signature of the label pair $\langle l_1, l_2\rangle$
$SL(\langle l_1, l_2\rangle), EL(< l_1, l_2 >)$	Start list and end list of the label pair $\langle l_1, l_2\rangle$
$SL(\langle l_1, l_2\rangle)\|_j$	Selections over the j-th elements in $SL(\langle l_1, l_2\rangle)\|_j$
$EL(\langle l_1, l_2\rangle)_k$	Selections over the k-th elements in $EL(\langle l_1, l_2\rangle)\|_k$
$M(e)$	Matches of query edge e
$M(Q)$	Matches of query graph Q

probe the candidates by signatures, to reduce the search space. In this section, we introduce our index structures formally. Then, in Sections 5–6, we discuss the online query processing algorithms. The index maintenance issue will be discussed in Section 7.

Definition 4 Given a vertex label l in graph G, $F^{-1}(l)$ is defined as $F^{-1}(l) = \{v|F(v) = l \wedge v \in V(G)\}$, where $V(G)$ denotes the set of vertices in graph G. Furthermore, we order all vertices in $F^{-1}(l)$ in the ascending order of their vertex IDs.

Definition 5 Given a directed edge $\overrightarrow{v_1v_2}$ in G, the vertex labels of v_1 and v_2 are l_1 and l_2, respectively. The *ID pair* of edge e is defined as (v_1, v_2); and the *label pair* of edge e is defined as $\langle l_1, l_2\rangle$.

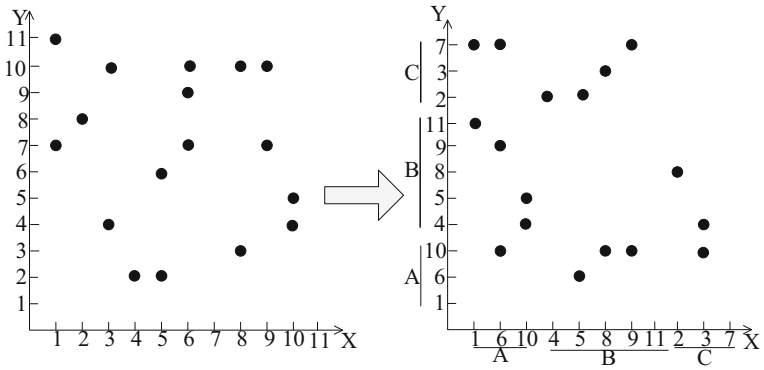
Given a label pair $\langle l_1, l_2\rangle$, $F^{-1}(\langle l_1, l_2\rangle) = \{(v_1, v_2)|F(v_1) = l_1 \wedge F(v_2) = l_2 \wedge \overrightarrow{v_1v_2} \in E(G)\}$ and $F^{-1}(\langle l_1, l_2\rangle)|_{l_1} = \{v_1|(v_1, v_2) \in F^{-1}(\langle l_1, l_2\rangle)\}$, where $F^{-1}(\langle l_1, l_2\rangle)$ denotes all directed edges with two ending points are l_1 and l_2, respectively.

Take graph G in Figure 1 for example. $F^{-1}(A) = \{1, 6, 10\}$ denotes all vertices whose labels are 'A'. Considering an edge $e = \overrightarrow{6, 9}$, its ID pair is denoted as (6, 9) and its label pair is denoted as $\langle A, B\rangle$. $F^{-1}(\langle A, B\rangle) = \{(6, 9), (10, 5), (1, 11), (10, 4)\}$. $F^{-1}(\langle A, B\rangle)|_A = \{1, 6, 10\}$ and $F^{-1}(\langle A, B\rangle)|_B = \{4, 5, 9, 11\}$.

For each edge e in G, according to its ID pair (Definition 5), we map it into a two-dimensional point in $\Re^2$. For example, given an edge $e = \overrightarrow{6, 9}$, its corresponding point in $\Re^2$ is (6, 9). Figure 2a shows the two-dimensional space $\Re^2$ that corresponds to graph G in Figure 1(a). The X and Y axis denote the starting and ending vertex of each edge, respectively. The ordering in X and Y axes are both in the increasing order of vertex IDs. In order to index $\Re^2$, we re-arrange X and Y axis in Definition 6.

Definition 6 Given two vertex v_i and v_j ($v_i \neq v_j$) in G, we say $v_i < v_j$ if and only if: 1) $F(v_i) < F(v_j)$, where $F(v_i)$ and $F(v_j)$ and the ordering for vertex labels are defined in Definition 1; or 2) $F(v_i) = F(v_j) \wedge (i < j)$.

According to the total order in Definition 6, we re-organize the vertex IDs in both X and Y axis, as shown in Figure 2b. We partition $\Re^2$ into different areas, and each



(a) $\Re^2$ (b) Re-organized $\Re^2$

Figure 2 Two-dimensional space $\Re^2$

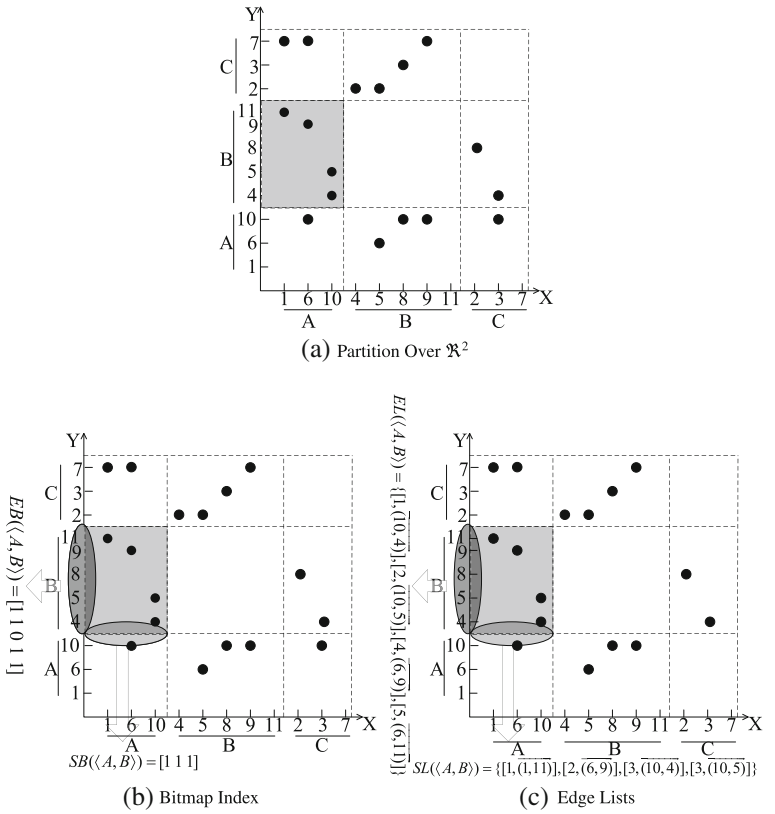
area corresponds to one *label pair* (Definition 5), as shown in Figure 3a. For example, the shaded area in Figure 3a corresponds to the label pair $\langle A, B\rangle$, and $F^{-1}(\langle A, B\rangle) = \{(6, 9), (10, 5), (1, 11), (10, 4)\}$. Note that, for ease of presentation, we use the terms "area" and "label pair" interchangeably in the following discussion.

In order to index points in each area, we propose the following indexing structures. Consider an area that corresponds to a label pair $\langle l_1, l_2\rangle$ in $\Re^2$. $F^{-1}(l_1) = \{v_1, v_2, ..., v_n\}$ and $F^{-1}(l_2) = \{v'_1, v'_2, ..., v'_m\}$. Assume that all vertices in $F^{-1}(l_1)$ and $F^{-1}(l_2)$ are in the increasing order of vertex IDs, respectively.

Definition 7 Given a label pair $\langle l_1, l_2\rangle$, $F^{-1}(l_1) = \{v_1, ..., v_m\}$ and $F^{-1}(l_2) = \{v'_1, ..., v'_n\}$, its *start signature* and *end signature* (denoted as $SB(\langle l_1, l_2\rangle)$ and $EB(\langle l_1, l_2\rangle)$) are defined as follows:

$SB(\langle l_1, l_2\rangle)$ is a length-m bit-string, denoted as $SB(\langle l_1, l_2\rangle) = [a_1, ..., a_m]$, where each bit a_i $(i = 1, .., m)$ corresponds to one vertex $v_i \in F^{-1}(l_1)$, and $\forall i \in [1, m]\ a_i = 1 \Leftrightarrow v_i \in F^{-1}(\langle l_1, l_2\rangle)|_{l_1}$.

$EB(\langle l_1, l_2\rangle)$ is a length-n bit-string, denoted as $EB(\langle l_1, l_2\rangle) = [b_1, ..., b_n]$, where each bit b_i $(i = 1, .., n)$ corresponds to one vertex $v'_i \in F^{-1}(l_2)$, and $\forall i \in [1, n]\ b_i = 1 \Leftrightarrow v'_i \in F^{-1}(\langle l_1, l_2\rangle)|_{l_2}$.



(a) Partition Over $\Re^2$

(b) Bitmap Index

(c) Edge Lists

Figure 3 Two-dimensional Space $\Re^2$

Let us recall the shaded area corresponding to label pair $\langle A, B\rangle$ in Figure 3a. Since $F^{-1}(A) = \{1, 6, 10\}$ and $F^{-1}(B) = \{4, 5, 8, 9, 11\}$, thus, $|SB(\langle A, B\rangle)| = 3$ and $|EB(\langle A, B\rangle)| = 5$. Since $F^{-1}(\langle A, B\rangle)|_A = \{1, 6, 10\}$, thus, $SB(\langle A, B\rangle) = [111]$. Since $F^{-1}(\langle A, B\rangle)|_B = \{4, 5, 9, 11\}$, thus, $EB(\langle A, B\rangle) = [11011]$.

Besides start and end signatures in $\Re^2$, for each area, we store all edges that are ranked by start and end vertex, respectively. Definition 8 defines the index structure.

Definition 8 Given a label pair $\langle l_1, l_2\rangle$, $F^{-1}(l_1) = \{v_1, ..., v_m\}$ and $F^{-1}(l_2) = \{v'_1, ..., v'_n\}$, its *start list* is defined as follows:

$$SL(\langle l_1, l_2\rangle) = \{[j, (\overrightarrow{v_j v_k})] | \overrightarrow{v_j v_k} \in F^{-1}(\langle l_1, l_2\rangle)\}$$

where $[j, (\overrightarrow{v_j v_k})]$ denotes one edge $\overrightarrow{v_j v_k} \in F^{-1}(\langle l_1, l_2\rangle)$ and j is called *start index*, which denotes the j-th bit that corresponds to v_j in $SB(\langle l_1, l_2\rangle)$.

Given a label pair $\langle l_1, l_2\rangle$, $F^{-1}(l_1) = \{v_1, ..., v_m\}$ and $F^{-1}(l_2) = \{v'_1, ..., v'_n\}$, its *end list* is defined as follows:

$$EL(\langle l_1, l_2\rangle) = \{[k, (\overrightarrow{v_j v_k})] | \overrightarrow{v_j v_k} \in F^{-1}(\langle l_1, l_2\rangle)\}$$

where $[k, (\overrightarrow{v_j v_k})]$ denotes one edge $\overrightarrow{v_j v_k} \in F^{-1}(\langle l_1, l_2\rangle)$ and k is called *end index*, which denotes the k-th bit that corresponds to v_k in $EB(\langle l_1, l_2\rangle)$.

There are four edges $\overrightarrow{6, 9}$, $\overrightarrow{10, 5}$, $\overrightarrow{1, 11}$, $\overrightarrow{10, 4}$ in the shaded area in Figure 3c. Since 1, 6 and 10 correspond to the 1-st, 2-nd and 3-rd bit in $SB(\langle A, B\rangle)$, respectively, thus, $SL(\langle A, B\rangle) = \{[1, (\overrightarrow{1, 11})], [2, (\overrightarrow{6, 9})], [3, (\overrightarrow{10, 5})], [3, (\overrightarrow{10, 4})]\}$. Analogously, $EL(\langle A, B\rangle) = \{[1, (\overrightarrow{10, 4})], [2, (\overrightarrow{10, 5})], [4, (\overrightarrow{6, 9})], [5, (\overrightarrow{1, 11})]\}$.

Definition 9 Given a start list $SL(\langle l_1, l_2\rangle)$ (or an end list $EL(\langle l_1, l_2\rangle)$), *selections* over the start list (denoted as $SL(\langle l_1, l_2\rangle)|_j$) and the end list (denoted as $EL(\langle l_1, l_2\rangle)|_k$) are defined as follows:

$$SL(\langle l_1, l_2\rangle)|_j = \{(v_j, v_k) | [j, \overrightarrow{v_j v_k}] \in SL(\langle l_1, l_2\rangle)\}$$
$$EL(\langle l_1, l_2\rangle)|_k = \{(v_j, v_k) | [k, \overrightarrow{v_j v_k}] \in EL(\langle l_1, l_2\rangle)\}$$

where j (k) denotes the bit position that corresponds to v_j (v_k) in $SB(\langle l_1, l_2\rangle)$ ($EB(\langle l_1, l_2\rangle)$).

For example, $SL(\langle A, B\rangle)|_3 = \{\overrightarrow{10, 4}, \overrightarrow{10, 5}\}$, since vertex 10 corresponds to the 3-rd bit in $SB(\langle A, B\rangle)$.

As discussed above, for each label pair $\langle l_1, l_2\rangle$, we assign it four associated data structures, that are start signature, end signature, start list and end list. We build a hash table as an indexing structure (called *HT* index), as shown in Figure 4, in which label pairs are keys and the associated data structures are values.

When we store all data structures on disk, we can build B+-tree over the keys (i.e., the label pairs) rather than the signatures and lists. Therefore, a basic unit in each B+-tree page is a "label pair". Each leaf entry in the B+-tree has a link to the corresponding start/end signatures and lists, as shown in Figure 5.

Furthermore, it is straightforward to extend our method to support multi-labelled graphs. For example, in Figure 6, we assume that vertex 11 has multiple labels as $\{A, B\}$ instead of a single label B. Then, edge $\overrightarrow{1, 11}$ occurs in the signatures and lists of both label pair $< A, B >$ and $< A, A >$. Specifically, we have the following edge signatures/lists in the two label pairs. The online query algorithm does not require revision to support the multi-labelled graphs.

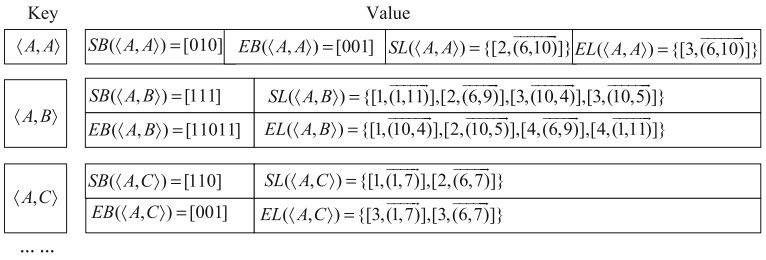


Figure 4 HT index

Last, in order to save the space cost of bitmap index, we propose to use the *compression version* of bitmap index, i.e., only recording the non-zero bit positions in start and end signatures. It is straightforward to know there are $2 \times |E(G)|$ non-zero bits in all start and end signatures in total, and there are $2 \times |E(G)|$ edges in all start and end lists in total. Therefore, we have the following theorem about the space cost of HT index.

Theorem 1 *The space complexity of HT index (in Figure 4) is* $O(|E(G)|)$.

Proof For each edge $e \in F^{-1}(\langle l_1, l_2\rangle)$ in G, there will be a non-zero bit in $SB(\langle l_1, l_2\rangle)$ and $EB(\langle l_1, l_2\rangle)$ and an element in $SL(\langle l_1, l_2\rangle)$ and $EL(\langle l_1, l_2\rangle)$, respectively. Because there are $|E(G)|$ edges, the space complexity of HT index is $O(|E(G)|)$. □

5 AEP-based query evaluation

In this section, we discuss how to evaluate a subgraph query over a large graph G. We first propose an edge join algorithm to answer an adjacent edge pair query (AEP query,

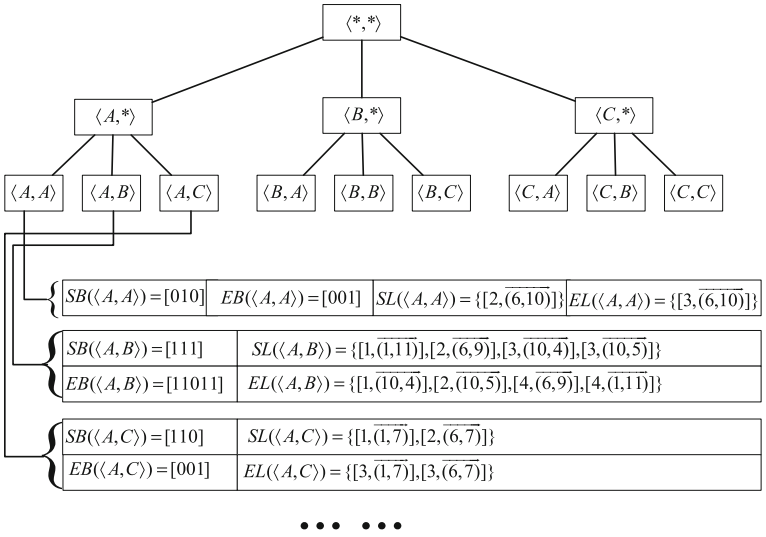
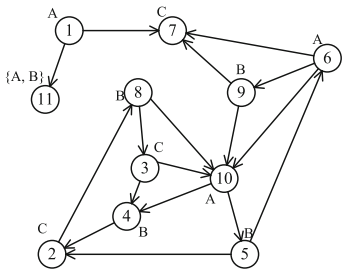


Figure 5 Index over B+-tree



(a) Data Graph G with Multiple Labels

$\langle A, A\rangle$	$SB(\langle A, A\rangle) = [1100]$	$SL(\langle A, A\rangle) = \{[1,\overrightarrow{(1,11)}],[2,\overrightarrow{(6,10)}]\}$
	$EB(\langle A, A\rangle) = [0011]$	$EL(\langle A, A\rangle) = \{[3,\overrightarrow{(6,10)}],[4,\overrightarrow{(1,11)}]\}$
$\langle A, B\rangle$	$SB(\langle A, B\rangle) = [1110]$	$SL(\langle A, B\rangle) = \{[1,\overrightarrow{(1,11)}],[2,\overrightarrow{(6,9)}],[3,\overrightarrow{(10,4)}],[3,\overrightarrow{(10,5)}]\}$
	$EB(\langle A, B\rangle) = [1101]$	$EL(\langle A, B\rangle) = \{[1,\overrightarrow{(10,4)}],[2,\overrightarrow{(10,5)}],[4,\overrightarrow{(6,9)}],[4,\overrightarrow{(1,11)}]\}$

(a) Signatures and Lists for Multiple Labels

Figure 6 Graph with Multiple Labels and Its Index

Definition 11) in Section 5.1. Given a general query Q with more than two edges, we find a set of AEP queries to cover Q, and find matches of Q by joining all AEP query results. In order to optimize query Q, we propose a cost model to guide finding AEPs in Q. The cost model and the general subgraph query algorithm will be discussed in Sections 5.2 and 5.3, respectively.

5.1 Edge join

Definition 10 Given two edges e_1 and e_2, e_1 is *adjacent* to e_2, if and only if e_1 and e_2 has one common vertex v. There are four cases for two adjacent edges e_1 and e_2.

1) e_1 and e_2 are called EE join if they share the same end point.
2) e_1 and e_2 are called SS join, if they share the same start point.
3) e_1 and e_2 are called SE join, if v is e_1's start point and v is also e_2's end point.
4) e_1 and e_2 are called ES join, if v is e_1's end point and v is also e_2's start point.

Definition 11 Given a query Q having two edges e_1 and e_2, if e_1 is adjacent to e_2, Q is called an **a**djacent **e**dge **p**air (AEP for short) query.

For ease of presentation, we only consider ES join in the following discussion, since other cases have the analogous query process. Let us recall an AEP query Q_1 in Figure 1, which has two adjacent edges $e_1 = \overrightarrow{u_1u_2}$ and $e_2 = \overrightarrow{u_2u_3}$ with one common vertex u_2. The label pairs of e_1 and e_2 are $\langle A, B\rangle$ and $\langle B, C\rangle$, respectively. Considering label pair $\langle A, B\rangle$, all edges in $F^{-1}(\langle A, B\rangle)$ are matches of e_1, i.e., $M(e_1) = F^{-1}(\langle A, B\rangle)$ (defined in Definition 5). Due to the same reason, $M(e_2) = F^{-1}(\langle B, C\rangle)$, wher $M(e_1)$ and $M(e_2)$ are both shown in Figure 7. The baseline algorithm is to perform a natural join $M(e_1) \bowtie M(e_2)$ based on the common column u_2. According to the index proposed in Section 4, we can find the edge lists that are ordered by u_2 for $M(e_1)$ and $M(e_2)$, respectively. Thus, we can perform a merge join to answer $M(e_1) \bowtie M(e_2)$. The join cost can be evaluated as follows:

$$\begin{aligned} Cost &= C_{IO} \times (|M(e_1)| + |M(e_2)|)/P_{disk} + C_{cpu} \times (Min(|M(e_1)|, |M(e_2)|) \\ &= 8 \times C_{IO}/P_{disk} + 4 \times C_{cpu} \end{aligned} \tag{1}$$

where C_{IO} is the average I/O cost for one disk page access, and $(|M(e_1)| + |M(e_2)|)/P_{disk}$ is the number of disk page accesses to loading $M(e_1)$ and $M(e_2)$ (i.e., $F^{-1}(\langle A, B\rangle)$ and $F^{-1}(\langle B, C\rangle)$) into memory, and C_{cpu} is the average CPU cost.

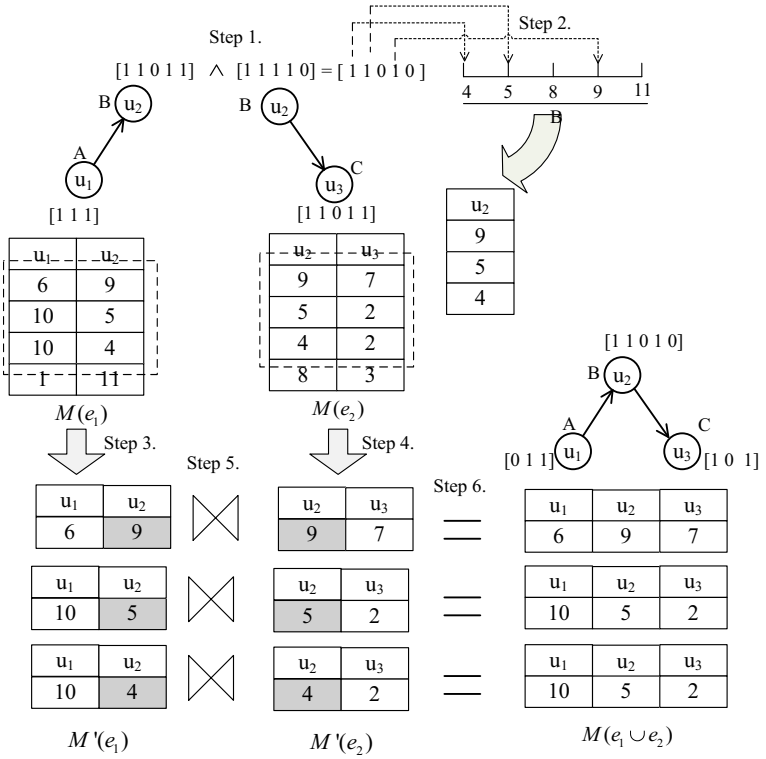


Figure 7 Edge join processing

In order to speed up query processing, we need to reduce the cost in (1). Actually, it is not necessary to load the whole $M(e_1)$ and $M(e_2)$ into memory, and then perform $M(e_1) \bowtie M(e_2)$. The intuition of our method is that we can utilize bitmap index proposed in Section 4 to reduce $M(e_1)$ and $M(e_2)$, respectively. We first illustrate the method by Figure 7 to demonstrate the join processing for query Q_1 in Figure 1. The label pairs of two adjacent edges in Q are $\langle A, B\rangle$ and $\langle B, C\rangle$. $EB(\langle A, B\rangle) \wedge SB(\langle B, C\rangle) = [11010]$ (Step 1 in Figure 7). The non-zero bit positions are $I_1 = 1$, $I_2 = 2$, $I_3 = 4$. Actually, these non-zero bit positions correspond to vertices 4, 5, 9 in G (Step 2). Then, we can find $M(e_1 \cup e_2)$ as follows (Steps 3-6 in Figure 7):

$$M(e_1 \cup e_2) = (EL(\langle A, B\rangle)|_1 \bowtie SL(\langle B, C\rangle)|_1) \cup (EL(\langle A, B\rangle)|_2 \bowtie SL(\langle B, C\rangle)|_2) \cup (EL(\langle A, B\rangle)|_4 \bowtie SL(\langle B, C\rangle)|_4)$$
$$= (10, 4) \bowtie (4, 2) \cup (10, 5) \bowtie (5, 2) \cup (6, 9) \bowtie (9, 7) = \{(10, 4, 2), (10, 5, 2), (6, 9, 7)\}$$

In this case, the join cost can be evaluated as follows:

$$\begin{aligned} Cost(e_1, e_2) &= \delta + C_{IO} \times (\textstyle\sum_{i=1}^{i=n} (|EL(e_1)|_{I_i}| + |SL(e_2)|_{I_i}|)/P_{disk} \\ &\quad + C_{cpu} \times \textstyle\sum_{i=1}^{i=n} Min(|EL(e_1)|_{I_i}|, |SL(e_2)|_{I_i}|) \\ &= \delta + 6 \times C_{IO} + 3 \times C_{cpu} \end{aligned} \tag{2}$$

where δ is the average cost for bitwise AND operation (Step 1. in Figure 7), which is so small to be neglected. Obviously, the cost in (2) is less than that in (1). The pseudo codes

for **E**dge **J**oin (EJ) algorithm are given in Algorithm 1. We can prove that the EJ algorithm satisfies *no-false-negative* requirement in Lemma 1.

Lemma 1 *Assume that there are two ES join edges e_1 and e_2, and their label pairs are $\langle l_1, l_2\rangle$ and $\langle l_2, l_3\rangle$, respectively.*
1) if $EB(\langle l_1, l_2\rangle) \wedge SB(\langle l_2, l_3\rangle)$ is a bit-string in which each bit is 0, then $M(e_1 \cup e_2)$ =NULL;
2) if $EB(\langle l_1, l_2\rangle) \wedge SB(\langle l_2, l_3\rangle)$ is a bit-string in which the I_i-th bit is 1, $i = 1, ..., n$, then $M(e_1 \cup e_2)$ can be evaluated by the following equation:

$$M(e_1 \cup e_2) = \bigcup_{i=1}^{i=n} (EL(\langle l_1, l_2\rangle)|_{I_i} \bowtie SL(\langle l_2, l_3\rangle)|_{I_i})$$

where $EL(\langle l_1, l_2\rangle)|_{I_i}$ and $SL(\langle l_2, l_3\rangle)|_{I_i}$ are defined in Definition 9.

Proof For each bit b_{I_i} in $EB(\langle l_1, l_2\rangle) \wedge SB(\langle l_2, l_3\rangle)$, assume that b_i corresponds to v_{I_i} in $F^{-1}(l_2)$, there are two cases:

1) if $b_{I_i} = 0$, then the I_i-th bit in $EB(\langle l_1, l_2\rangle)$ is 0 or the I_i-th bit in $SB(\langle l_2, l_3\rangle)$ is 0. Therefore, there exists no $\overrightarrow{uv_{I_i}}$ or $\overrightarrow{v_{I_i}w}$ where $u \in F^{-1}(l_1)$ and $w \in F^{-1}(l_3)$. Therefore, $EL(\langle l_1, l_2\rangle)|_{I_i} \bowtie SL(\langle l_2, l_3\rangle)|_{I_i} = \Phi$;
2) if $b_{I_i} = 1$, then the I_i-th bit in $EB(\langle l_1, l_2\rangle)$ is 1 and the I_i-th bit in $SB(\langle l_2, l_3\rangle)$ is 1. Hence, there exist both $\overrightarrow{uv_{I_i}}$ and $\overrightarrow{v_{I_i}w}$ where $u \in F^{-1}(l_1)$ and $w \in F^{-1}(l_3)$. Thus, all edges $\overrightarrow{uv_{I_i}}$ and $\overrightarrow{v_{I_i}w}$ satisfy $e_1 \cup e_2$. Therefore, $M(e_1 \cup e_2) \supseteq EL(\langle l_1, l_2\rangle)|_{I_i} \bowtie SL(\langle l_2, l_3\rangle)|_{I_i}$;

On the basis of the above, we can conclude that $M(e_1 \cup e_2) = \bigcup_{i=1}^{i=n}(EL(\langle l_1, l_2\rangle)|_{I_i} \bowtie SL(\langle l_2, l_3\rangle)|_{I_i})$ □

Algorithm 1 **E**dge **J**oin (EJ) Algorithm

Require: **Input**: Given an AEP query p with two adjacent edges $e_1 = \overrightarrow{u_1u_2}$ and $e_2 = \overrightarrow{u_2u_3}$. Their label pairs are $\langle l_1, l_2\rangle$ and $\langle l_2, l_3\rangle$, respectively. Assume that e_1 and e_2 are ES joined.
Output: $M(e_1 \cup e_2)$.

```
1: According to HT index, we load EB(⟨l1, l2⟩) and SB(⟨l2, l3⟩) into memory.
2: Let b = EB(⟨l1, l2⟩) ∧ SB(⟨l2, l3⟩).
3: if b is a bit-string in which all bits are 0 then
4:     M(e1 ∪ e2) = ϕ.
5: else
6:     for i=1,...,|b| do
7:         if b[i] = 0 then
8:             continue
9:         else
10:            M(e1 ∪ e2) = M(e1 ∪ e2) ∪ (EL(⟨l1, l2⟩)|i ⋈ SL(⟨l2, l3⟩)|i)
11: Return M(e1 ∪ e2).
```

As discussed early, due to the clustered B^+-tree in the start and end lists, we can save I/O cost in the selections over these lists and employ the merge join in Line 10 of Algorithm 1.

5.2 Cost estimation

In this subsection, we propose a method to estimate the join cost in EJ algorithm, which will be used in Section 5.3 to answer a subgraph query. Let us recall the cost model in (2). It is easy to estimate δ, P_{disk} and C_{IO} and C_{CPU} from the collected statistics of query data. The key issue is how to estimate $|EL(e_1)|_{I_i}|$ and $|SL(e_2)|_{I_i}|$. In order to address this

problem, we propose a histogram-based approach. For each label pair $\langle l_1, l_2\rangle$, we build two histograms, denoted as $SH(\langle l_1, l_2\rangle)$ and $EH(\langle l_1, l_2\rangle)$.

Definition 12 Given a label pair $\langle l_1, l_2\rangle$, the *start histogram* for $\langle l_1, l_2\rangle$ is a length-n number array, denoted as $SH(\langle l_1, l_2\rangle) = [h_1, ..., h_n]$, and each number h_i $(i = 1, ..., n)$ corresponds to one vertex v_i in $F^{-1}(l_1)$, and $\forall i \in [1, n]$, $h_i = |SL(\langle l_1, l_2\rangle)|_i|$.

The *end histogram* for $\langle l_1, l_2\rangle$ is a length-m number array, denoted as $EH(\langle l_1, l_2\rangle) = [h_1, ..., h_m]$, and each bit h_i $(i = 1, ..., m)$ corresponds to one vertex v_i in $F^{-1}(l_2)$, and $\forall i \in [1, m]$, $h_i = |EL(\langle l_1, l_2\rangle)|_i|$.

In order to estimate $Cost(e_1 \cup e_2)$ for query Q_1 in Figure 7, we first compute $EB(e_1) \wedge SB(e_2) = [11010]$, in which there are 3 non-zero bits, that are the 1-st, 2-nd and 4-th positions in $EB(e_1)$ and $SB(e_2)$, respectively. According to the $EH(e_1)$ and $SH(e_2)$, it is straightforward to estimate $|EL(e_1)|_i|$ and $|SL(e_2)|_i|$, where $i \in \{1, 2, 4\}$. Finally, we can estimate $Cost(e_1 \cup e_2)$ by (2).

We list the pseudo codes for the **c**ost **e**stimation (CE) algorithm in Algorithm 2. Given two ES join edges $e_1 = \overrightarrow{u_1u_2}$ and $e_2 = \overrightarrow{u_2u_3}$, their label pairs are $\langle l_1, l_2\rangle$ and $\langle l_2, l_3\rangle$. We first compute $r = EB(e_1) \wedge SB(e_2)$. For i-th bit in r $(i = 1, ..., |r|)$, if it is a non-zero bit, we can estimate $|EL(e_1)|_i|$ and $|SL(e_2)|_i|$ according to the i-th element in $EH(e_1)$ and $SH(e_2)$, respectively. Finally, we estimate the join cost by (2).

Algorithm 2 **C**ost **E**stimation (CE) Algorithm

Require: **Input**: Given an AEP query p with two adjacent edges $e_1 = \overrightarrow{u_1u_2}$ and $e_2 = \overrightarrow{u_2u_3}$. Their label pairs are $\langle l_1, l_2\rangle$ and $\langle l_2, l_3\rangle$, respectively. Assume that e_1 and e_2 are ES joined.
Output: cost estimation $Cost$.

```
1: We load EB(⟨l1, l2⟩) and SB(⟨l2, l3⟩) into memory.
2: Let b = EB(⟨l1, l2⟩) ∧ SB(⟨l2, l3⟩).
3: for i=1,...,|b| do
4:     Cost = Cost + C_IO × (EH[i] + SH[i])/P_disk + C_CPU × (EH[i] × SH[i])
5: Return Cost
```

5.3 AEP-based query algorithm

In order to answer a subgraph query Q ($|E(Q)| > 2$), we find a set of AEP (Definition 11) queries to *cover* Q (see Definition 13). Then, for each AEP $(e_i \cup e_j)$, we employ EJ algorithm to find $M(e_{i_1} \cup e_{i_2})$. For ease of presentation, we use p_i to denote an AEP $(e_{i_1} \cup e_{i_2})$. For any two distinct adjacent edge pairs p_1 and p_2 in S, there are only three topological relations, *disjointed*, *one-vertex sharing* and *two-vertex sharing*.

Definition 13 Given a set of AEP, denoted as $AS = \{p_i = (e_{i_1} \cup e_{i_2})\}$ in Q, we say that AS *covers* Q if and only if $\bigcup\{p_i\} = Q$.

We say that AS is a *minimal cover* over Q, if and only if AS satisfies the following two conditions: (1)AS covers Q; and (2)Removing any AEP from AS will lead that AS cannot cover Q.

Considering each AEP p_i, we can estimate $Cost(p_i)$ by (2). As we know, in order to answer a subgraph query Q, we need to answer a set of AEP queries. The cost of answering Q can be evaluated by the sum of all edge joins. Thus, we use $Cost(AS)$ in (3) to estimate the cost for answering Q.

Definition 14 Given a query Q, AS is called *the minimum cover* over Q if and only if $Cost(AS)$ is the smallest over all minimal covers over Q.

$$Cost(AS) = \sum Cost(p_i) = \sum Cost(e_{i_1} \cup e_{i_2}) \tag{3}$$

where $p_i = (e_{i_1} \cup e_{i_2}) \in AS$ and $Cost(e_{i_1} \cup e_{i_2})$ is defined in (2).

Given a query Q, there may exist more than one minimal cover over Q. In order to optimize subgraph query processing, we need to find the *minimum cover* (Definition 14). Luckily, we can reduce the minimum cover to the edge cover problem. The edge cover problem [21] is given in the following definition. Since the edge cover problem can be solved in a polynomial time algorithm [21], thus, finding the minimum query cover in AEP can also solved in polynomial time.

Lemma 2 *Finding the minimum cover based on AEPs can be reduced to the edge cover problem.*

Proof We can reduce an AEP-based minimum query cover problem to a minimum edge cover problem as follows. Here, we construct an weighted undirected graph $G^* = (V^*, E^*, W)$ as follows. Each edge e_i in $E(Q)$ maps to a vertex v_i^* in V^*. If an AEP p_i consists of two edges e_i and e_j of $E(Q)$, there is an edge e^* in E^* between v_i^* and v_j^*. The weight of e^* is the estimated cost of p_i. It is straightforward to know each instance of the AEP-based query cover problem can be computed by invoking the edge cover algorithm. □

Theorem 2 *Finding the minimum query cover in AEP-based algorithm can be solved in polynomial time.*

Proof The edge cover problem can be solved in polynomial time [21]. According to Lemma 2, we know the theorem holds. □

According to Theorem 2, we propose a polynomial AEP-based Query algorithm as Algorithm 3. First, we use the edge cover algorithm to find the minimum cover AS of Q. Then, we employ EJ algorithm to find matches of all AEPs in AS and join these matches together.

Algorithm 3 AEP-based Query Algorithm

Require: **Input**: a query Q, a data graph G and index HT.
Output: Matches set of Q in G, denoted as $M(Q)$.
1: Construct an weighted undirected graph G^* based on all AEPs in Q. // Lemma 2.
2: Find the minimum edge cover ES in G^*. // [22].
3: According to ES, find the minimum cover AS based on AEPs.
4: Select the AEP p in AS with the smallest estimation cost.
5: Employ EJ algorithm (Algorithm 1) to find matches $M(p)$.
6: $M(Q) = M(p)$.
7: $AS = AS - p$.
8: **while** $AS \neq \emptyset$ **do**
9: Select the AEP p' (in AS) with the smallest estimation cost
10: Employ EJ algorithm to find $M(p')$.
11: $M(Q) = M(Q) \bowtie M(p')$.
12: Return $M(Q)$.

Let us consider an example in Figure 8. Given a query graph Q_2 in Figure 1(b), it has three candidate AEPs, i.e., $S = \{p_1, p_2, p_3\}$, where $p_1 = \langle A, B\rangle \wedge \langle B, A\rangle$, $p_2 = \langle A, B\rangle \wedge \langle B, C\rangle$, $p_3 = \langle B, A\rangle \wedge \langle B, C\rangle$. Since there are three edges in Q, we introduce the three corresponding vertices in G^*. There is an edge between two vertices in G^* if and only if their corresponding edges (in Q) are in one AEP. Each edge in G^* has a cost, which equals to the cost of each AEP. We find the minimum edge cover in G^*. Based on that, we can find the minimum query cover as $\{p_1, p_2\}$. Then, we can find all matches of p_1 and p_2 are $M(p_1) = \{(6, 9, 10), (10, 5, 6)\}$ and $M(p_2) = \{(6, 9, 7), (10, 5, 2), (10, 4, 2)\}$. Finally, we can join $M(p_1)$ and $M(p_2)$ based on the common vertices u_1 and u_2, i.e., $M(Q) = M(p_1) \bowtie_{u_1,u_2} M(p_2) = \{(6, 9, 10, 7), (10, 5, 6, 2)\}$.

Theorem 3 *Our AEP algorithm can yield correct subgraph matches.*

Proof Given query graph Q and data graph G, supposed that AS is the minimum cover based on AEPs, we only need to prove that our algorithm does not lead to false positive and negative results.

Supposed that G' is a match that we find by using AEP algorithm, for each $v \in V(G')$ and $e \in E(G')$, it belongs to at least one AEP in AS. Hence, for each v and e, we at least employ EJ algorithm once, which can guarantee that v and e have corresponding matches in G. Thus, there is no false positive results.

For each match G'' of Q in G, it can be covered by the same edge pairs set to AS, so we must be able to get it by joining the result of each edge pair in AS. Thus, there is no negative results.

On the basis of the above, we can know that our algorithm is correct. □

6 Optimization

In this section, we propose two optimization methods to speed up subgraph query processing.

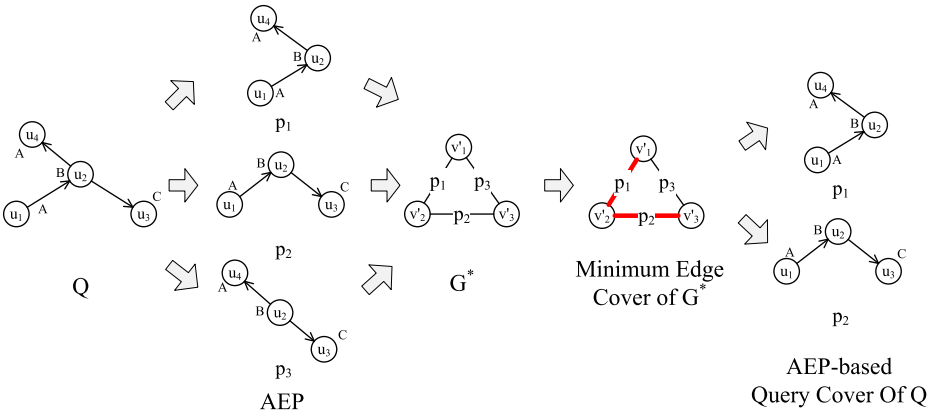


Figure 8 Finding the Edge Cover

6.1 SSP-based query evaluation

In Algorithm 3, we find a set of AEP queries to cover query Q. If some vertices in Q have high degrees, we have a large number of AEP queries. So, it will be expensive to select a minimal cover over Q. Furthermore, if query graph size is large, the number of join steps in Line 8 in Algorithm 3 is also large. Let us recall query Q_2 in Figure 1(b). We need to join two AEP queries to answer Q_2, however, we can join all edges in Q_2 in one step, since they have one common vertex u_2.

Definition 15 Given a query Q having m edges, these edges have one common vertex u (called *center*), Q is called a *star-style pattern (SSP)* query.

We firstly illustrate SSP query process by Q_2. For example, we have three edges in Q_2 with center u_2. For edge $e_1 = \overrightarrow{u_2u_4}$ and $e_2 = \overrightarrow{u_2u_3}$, u_2 is a start point in the two edges, respectively. For edge $e_3 = \overrightarrow{u_1u_2}$, u_2 is an end point. Therefore, we join the corresponding signatures together, i.e., $SB(e_1) \wedge SB(e_2) \wedge EB(e_3) = SB(\langle B, A\rangle) \wedge SB(\langle B, C\rangle) \wedge EB(\langle A, B\rangle) = [01010]$. Since the second and fourth bits are '1', these bits correspond to vertices 5 and 9, respectively. Finally, we perform the following join process in (4). Figure 9 illustrates the details about SSP query processing.

$$\begin{aligned} M(e_1 \cup e_2 \cup e_3) = (SL(e_1)|_2 \bowtie SL(e_2)|_2 \bowtie EL(e_3)|_2) \\ \cup (SL(e_1)|_4 \bowtie SL(e_2)|_4 \bowtie EL(e_3)|_4) \end{aligned} \tag{4}$$

Algorithm 4 *E*xtended *E*dge *J*oin(E-EJ)

Require: **Input**: a start-style pattern query Q having m edges e_i ($i = 1, ..., n$) with center u.
Output: Matches set of Q in G, denoted as $M(Q)$.

1: $r = XB(e_1) \wedge XB(e_2) \wedge ... \wedge XB(e_m)$, where XB denotes SB or EB, which depends on the center position in e_i, $i = 1, ..., m$.
2: **for** each non-zero bit I_i in r **do**
3: $M(Q) = M(Q) \bigcup (XL(e_1)|_{I_i} \bowtie |XL(e_2)|_{I_i} ... \bowtie XL(e_m)|_{I_i})$, where XL denotes SL or EL, which depends on the center position on each edge e_i.
4: Report $M(Q)$.

In order to answer SSP queries, an **e**xtended **e**dge **j**oin (E-EJ) algorithm is proposed in Algorithm 4. Consider a SSP query Q having m edges e_i ($i = 1, .., m$) with center u. According to the center position (start or end points) in each edge e_i, we join the corresponding signatures (start or end signatures) together. Assume that are n non-zero bits in the join result, denoted as I_i, $i = 1, ..., n$. We use (5) to find the answers for query Q.

$$M(Q) = \bigcup_{i=1}^{i=n} (XL(e_1)|_{I_i} \bowtie |XL(e_2)|_{I_i} ... \bowtie XL(e_m)|_{I_i}) \tag{5}$$

where XL denotes SL or EL, which depends on the center position (start or end points) in each edge e_i.

Given a SSP query Q having m edges e_j ($j = 1, ..., m$) with the center u, the cost of SSP query can be modeled by (6). Obviously, we can still employ the histogram proposed in Section 2 to estimate (6).

$$\begin{aligned} Cost(Q) = \delta + C_{IO} \times (\textstyle\sum_{i=1}^{i=n} (\sum_{j=1}^{j=m} (|XL(e_j)|_{I_i}|)))/P_{disk} + \\ C_{cpu} \times (\textstyle\sum_{i=1}^{i=n} (Min_{j=1}^{j=m} (|XL(e_j)|_{I_i}|))) \end{aligned} \tag{6}$$

Then, we need to find the the minimum query cover with SSPs. Here, we propose a heuristic solution to address this problem. Initially, we set $Q' = \phi$. We firstly find a star-style pattern s_1 (in Q) with the minimal estimated cost. Then, we can employ Algorithm 4

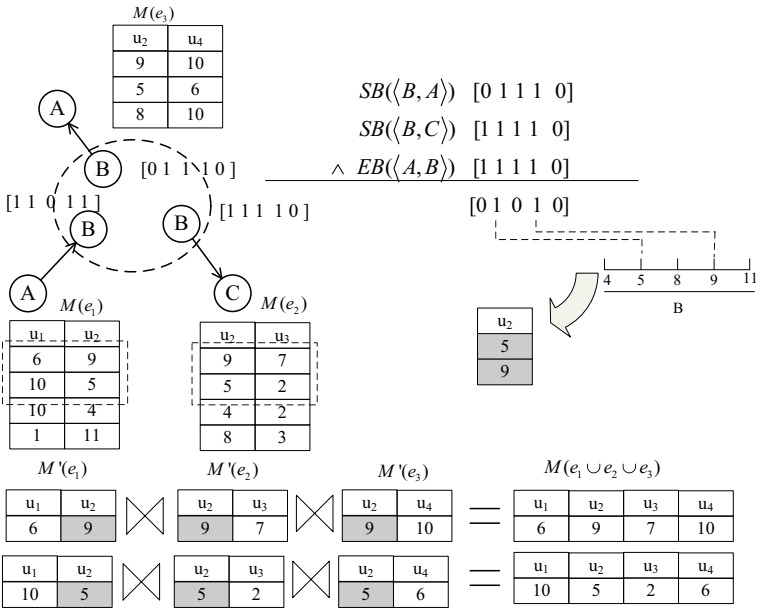


Figure 9 Extended edge join processing

to find matches for s_1, i.e., $M(s_1)$. We insert s_1 into Q' and remove it from Q. Then, we try to find some edges $e(\overrightarrow{u_i u_j})$ in Q, where u_i and u_j are already in s_1. In this case, we refer these edges as *backward edges*. In order to find matches for $s_1 \cup e$, denoted as $M(S_1 \cup e)$, we can scan $M(s_1)$ to filter some matches that are against the edge condition $e(\overrightarrow{u_i u_j})$. Then, we insert all backward edges in s_1. Next, we find another SSP query s_i that is adjacent to Q' and has the minimal estimation cost. Then, we iterate the above process until $Q' = Q$. Finally, we report $M(Q)$. Algorithm 5 shows our SSP-based Query Algorithm.

Algorithm 5 SSP-based Query Algorithm

Require: **Input**: A Query Q and a data graph G.
Output: Matches set of Q in G, denoted as $M(Q)$.
1: Initialize $Q' = \phi$.
2: **while** $Q' != Q$ **do**
3: Finding a SSP subquery S_1 with center u in Q, which is adjacent to Q' and has the minimal cost estimation.
4: Employ Algorithm 4 to find matches for S, denoted as $M(S_1)$.
5: $Q' = Q' \cup S_1$.
6: Find some edges $e(u_i u_j)$ in Q, where u_i and u_j are in Q'.
7: Based on edge connections $e(u_i u_j)$, we scan $M(S_1)$ to find matches for $M(S_1 \cup e)$.
8: Insert e into S_1.
9: $M(Q')$=$M(Q') \ltimes M(S_1)$.
10: Report $M(Q)$.

Theorem 4 *Our SSP algorithm can yield correct subgraph matches.*

Proof Its proof is similar as the proof of Theorem 3. □

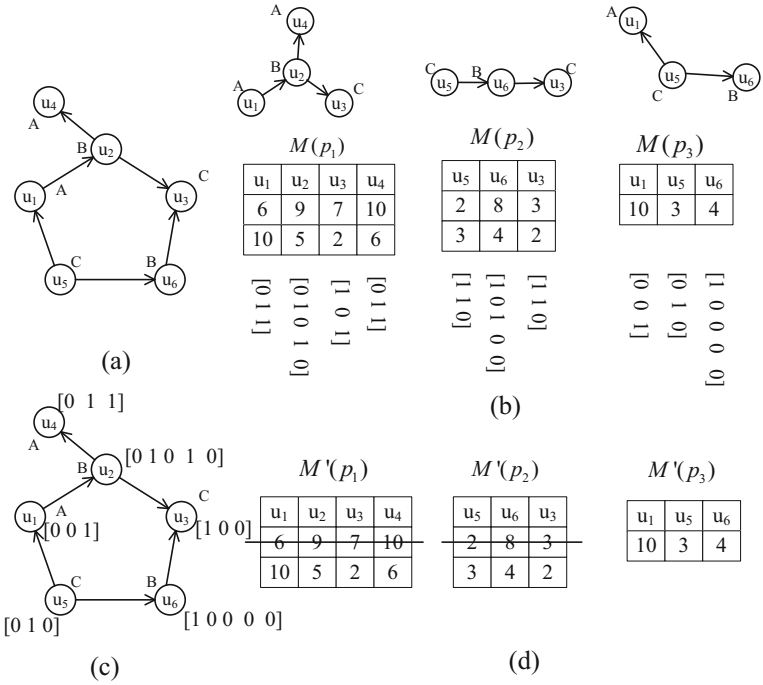


Figure 10 Reducing Intermediate Result Size

6.2 Reducing intermediate results

In both AEP and SSP, we need to perform a series of two-way natural joins to find matches of query Q. If the intermediate result size is large, it will affect the query performance. In our experiments, we find that some queries have a few matches over graph G, but, intermediate result sizes are very large. In order to address this issue, we propose to utilize "signature" to reduce intermediate result size. Note that, the following method can work for both AEP and SSP-based solution. For ease of presentation, we only use SSP as an example to illustrate how the optimization method works.

Given a query graph Q in Figure 10a, we can find three SSP queries (p_1, p_2 and p_3) to cover Q. We first find matches for these SSP queries by Algorithm 4, respectively. Instead of performing natural joins directly, we can first perform "semi-join" by signatures. For each p_i, according to $M(p_i)$, we can get a signature for each vertex in p_i, as shown in Figure 10b. Take vertex u_1 in $M(p_1)$ for example. u_1's label is 'A' and there are two vertices 6 and 10 that corresponds to u_1. Thus, we can get a signature [0 1 1] for u_1 in p_1. Similarly, vertex u_1 is also in $M(p_3)$. According to $M(p_3)$, u_1's signature is [0 0 1].

For each vertex u in query Q, if u occur in different SSP queries p_i, we perform bitwise ADD operations over signatures associated with u in different p_i, as shown in Figure 10c. According to the signatures, we can remove some matches from $M(p_i)$. For example, u_1 appears in both p_1 and p_2. Thus, according to their corresponding signatures, [0 1 1] $\cap$ [0 0 1] = [0 0 1]. Since u_1's signature is [0 0 1], we can remove (6, 9, 7, 10) from $M(p_1)$ directly. In this way, we can get "shrunk" lists $M(p_i)$. Finally, we perform a series of two-way natural joins over these shrunk lists, which leads to less intermediate results.

Because of the signature-based pruning techniques, the intermediate result size is so small that we can maintain the intermediate results in memory in our experiments. If the intermediate result size is too large to be cached in memory, we have to flush some of them to disk. Then, we perform disk-based natural join algorithms over these intermediate results. This is a well-studied problem in RDBMS, such as hash join and merge join algorithms. We omit discussion about this tangential issue in this paper.

7 Maintenance

In some real applications, the underlying data graph is not static. In this section, we address index maintenance issues to support online updates over graph databases.

7.1 Insertion

Assume that we insert an edge $\overrightarrow{v_1v_2}$ into graph G and its label pair is $\langle l_1, l_2\rangle$. There are five different cases to be considered.

7.1.1 $v_1 \in V(G) \wedge v_2 \in V(G)$

When we insert $e = \overrightarrow{v_1v_2}$ (its label pair is $\langle l_1, l_2\rangle$) into G, if $v_1 \in V(G) \wedge v_2 \in V(G)$, it means that we do not introduce any new vertex ID. Therefore, we only need to update start (end) signatures and start (end) lists for label pair $\langle l_1, l_2\rangle$. As discussed early, each bit in $SB(\langle l_1, l_2\rangle)$ (and $EB(\langle l_1, l_2\rangle)$) corresponds to one vertex in $F^{-1}(l_1)$ (and $F^{-1}(l_2)$, Definition 4). Assume that v_1 corresponds to the i-th bit in $SB(\langle l_1, l_2\rangle)$ and v_2 corresponds to the j-th bit in $EB(\langle l_1, l_2\rangle)$. We set the i-th bit in $SB(\langle l_1, l_2\rangle)$ to be '1' and set j-th bit in $EB(\langle l_1, l_2\rangle)$ to be '1'. We also insert edge $[i, \overrightarrow{v_iv_j}]$ into $SL(\langle l_1, l_2\rangle)$ and $EL(\langle l_1, l_2\rangle)$, respectively.

Cost Analysis. As discussed above, we first need to update two signatures $SB(\langle l_1, l_2\rangle)$ and $EB(\langle l_1, l_2\rangle)$. Since the length of a signature is equal to the number of vertices with the same label, the complexity of updating the signatures is $O(|V|/|L|)$. We also need to insert two edges into two lists $SL(\langle l_1, l_2\rangle)$ and $EL(\langle l_1, l_2\rangle)$, respectively. Due to the B$^+$-trees in these lists, the time complexity for insertion is $O(log|F^{-1}(\langle l_1, l_2\rangle)|)$, where $F^{-1}(\langle l_1, l_2\rangle)$ is defined in Definition 5. Therefore, the total time complexity is $O(log|F^{-1}(\langle l_1, l_2\rangle)|)$.

7.1.2 $v_1 \notin V(G) \wedge l_1 \in L_V$

When we insert $e = \overrightarrow{v_1v_2}$ (its label pair is $\langle l_1, l_2\rangle$) into G, $v_1 \notin V(G) \wedge l_1 \in L_V$ means that we introduce a new vertex ID but not a new vertex label. Firstly, we need to update start and end signatures for all label pairs $\langle l_1, X\rangle$ and $\langle X, l_1\rangle$, where $X \in L_V$. Considering a label pair $\langle l_1, X\rangle$ (and $\langle X, l_1\rangle$), we need to enlarge $SB(\langle l_1, X\rangle)$ (and $EB(\langle X, l_1\rangle)$) by one bit (initializing to be '0') that corresponds to vertex ID v_1. Then, we employ the method in Section 7.1.1, i.e., updating $SB(\langle l_1, l_2\rangle)$ and $SB(\langle l_1, l_2\rangle)$, and inserting $\overrightarrow{v_1v_2}$ into $SL(\langle l_1, l_2\rangle)$ and $SL(\langle l_1, l_2\rangle)$, respectively.

Cost Analysis. Since there are $2 \times |L_V|$ label pairs to be considered in the first step, thus, the time complexity is $O(|L_V|)$. We employ the method in Section 7.1.1 in the second step, thus, the time complexity is $O(log|F^{-1}(l_1, l_2)|)$. Therefore, the total complexity is

$O(|L_V| + log|F^{-1}(l_1, l_2)|)$. Actually, if we adopt the compression version of bitmap index (i.e, only recoding the positions for bit '1'), the first step can be ignored.

7.1.3 $v_2 \notin V(G) \wedge l_2 \in L_V$

It is analogous to Section 7.1.2.

7.1.4 $v_1 \notin V(G) \wedge l_1 \notin L_V$

When we insert $e = \overrightarrow{v_1 v_2}$ (its label pair is $\langle l_1, l_2 \rangle$) into G, $v_1 \notin V(G)$ and $l_1 \notin L_V$ means that we introduce a new vertex ID having a new vertex label l_1. Firstly, we introduce $2 \times |L_V|$ new label pairs $\langle l_1, X \rangle$ and $\langle X, l_1 \rangle$, where $X \in L_V$. For each label pair, we assign it start (and end) signatures and start (and end) lists. Initially, all elements in start (and end) signatures are '0' and all start (and end) lists are empty. Then, we employ the method in Section 7.1.1 to update $SB(l_1, l_2)$ and $EB(l_1, l_2)$, and insert $\overrightarrow{v_1 v_2}$ into $SL(l_1, l_2)$ and $EL(l_1, l_2)$, respectively. If we adopt the compression version of the bitmap index, we only need to introduce one new label pair $\langle l_1, l_2 \rangle$, and then employ the method in Section 7.1.1.

Cost Analysis. We need to introduce $2 \times |L_V|$ new label pairs in the first step, thus, the time complexity is $O(|L_V|)$. We employ the method in Section 7.1.1 in the second step, thus, the time complexity is $O(log|F^{-1}(l_1, l_2)|)$. Therefore, the total complexity is $O(|L_V| + |F^{-1}(l_1, l_2)|)$.

7.1.5 $v_2 \notin V(G) \wedge l_2 \notin L_V$

It is analogous to Section 7.1.4.

7.2 Deletion

Assume that we delete an edge $\overrightarrow{v_1 v_2}$ from graph G and its label pair is $\langle l_1, l_2 \rangle$. Firstly, we delete edge $\overrightarrow{v_1 v_2}$ from $SL(\langle l_1, l_2 \rangle)$ and $EL(\langle l_1, l_2 \rangle)$, respectively. Assume that v_1 corresponds to the i-th bit in $SB(\langle l_1, l_2 \rangle)$, and v_2 corresponds to the j-th bit in $EB(\langle l_1, l_2 \rangle)$. After deletion, if $SL(\langle l_1, l_2 \rangle)|_i = \phi$, we update the i-th bit in $SB(\langle l_1, l_2 \rangle)$ to be '0'. Similarly, if $EL(\langle l_1, l_2 \rangle)|_j = \phi$, we update the j-th bit in $EB(\langle l_1, l_2 \rangle)$ to be '0'.

Cost Analysis. Due to B^+-trees in $SL(\langle l_1, l_2 \rangle)$ and $EL(\langle l_1, l_2 \rangle)$, the time complexity of deletion is $O(log|F^{-1}(\langle l_1, l_2 \rangle)|)$.

8 Handling undirected graphs

Note that, all the techniques mentioned above are designed for directed graphs. Actually, our method can also be extended to support subgraph query over a large undirected graph. The intuition behind the extension is that we can convert an undirected graph G and query Q into the corresponding directed graph G^* and query Q^* without introducing false negative and positive results. Specifically, for any graph G(or Q) $V(G^*) = V(G)$, $L_V(G^*) = L_V(G)$ and the labeling function of G^* is equal to the labeling function of G. Besides, for each edge in G(or Q), if $F(v_1) < F(v_2)$ (or $F(v_2) < F(v_1)$, according to the alphabetical order), where F is the labeling function of G(or Q), then the directed edge $\overrightarrow{v_1 v_2}$ ($\overrightarrow{v_2 v_1}$) is

introduced into G^*(or Q^*); if $F(v_1) = F(v_2)$, then we introduce both directed edges $\overrightarrow{v_1v_2}$ and $\overrightarrow{v_2v_1}$ into G^*(or Q^*). There is an example in the following.

Given an undirected graph G, $V(G^*)$, $L_V(G^*)$ and labeling function F^* of G^* are the same with $V(G)$, $L_V(G)$ and the labeling function F of G. For example, because $F(v_1) = A < F(v_7) = C$, we introduce a directed edge $\overrightarrow{v_1v_7}$ into G^*, as shown in Figure 11. If $F(v_1) = F(v_2)$, then we introduce both directed edges $\overrightarrow{v_1v_2}$ and $\overrightarrow{v_2v_1}$ into G^*(or Q^*). For example, because $F(v_6) = A = F(v_{10}) = A$, we introduce directed edges $\overrightarrow{v_6v_{10}}$ and $\overrightarrow{v_{10}v_6}$ into G^*. Figure 11b shows an directed graph G^* that corresponds to G. Given an undirected query graph Q, we can also get its directed version Q^*, shown in Figure 12.

Considering undirected graph G and Q, we can find the match of Q in G (denoted as G'), shown in Figure 12a. According to our method, we can also find the match of Q^* (denoted as G'') in G^*. Note that G' and G'' have the exactly same vertex IDs. Thus, our extension method does not lead to any false negative or false positive result, as proved in Theorem 5.

Theorem 5 *The extension method does not lead to false positive and negative results.*

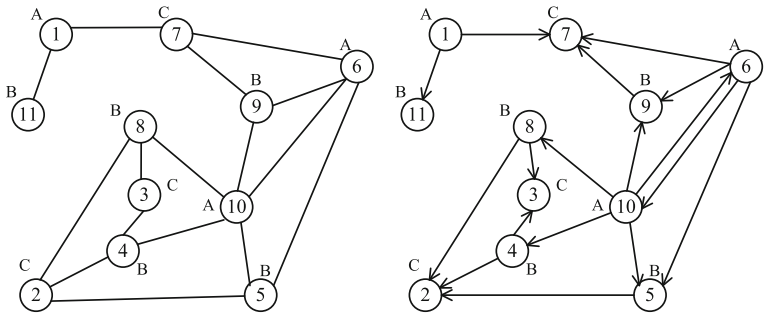
Proof Obviously, for each undirected graph G, there is one and only one corresponding directed graph G^* and different undirected graphs are mapped to different directed graphs. This means that the function according to our extension method is an injective function. Moreover, the function from Q to its match in G is a bijection function

Hence, according to the properties of relations, we know that for each match of Q in G, there exist a corresponding match of Q^* in G^* and for each match of Q^* in G^*, there exist a corresponding match of Q in G.

On the basis of the above, we can conclude that our extension method does not lead to false positive and negative results. □

9 Experiments

In this section, we evaluate our methods AEP (adjacent edge pair query) and SSP (star-style pattern query) over both synthetic and real data sets, and compare them with some state-of-the-art algorithms, such as GADDI [37], Nova [42], SPath [39] and TurboISO [12]. Our methods have been implemented using standard C++. The experiments are conducted on a P4 2.0GHz machine with 2Gbytes RAM running Linux. Furthermore, GADDI and Nova's



(a) Undirected Graph G (b) Directed Graph G^*

Figure 11 Example of extending undirected graph to directed graph

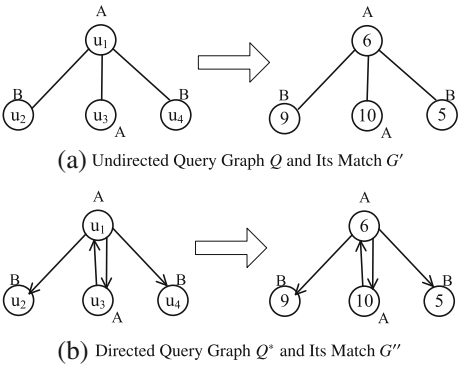


Figure 12 Converting undirected query graph to directed query graph

softwares are provided by authors. So far, the softwares of SPath and TurboISO have not been publicly-available, thus we use best-effort re-implementation according to [12, 39]. Since all competitors are designed for undirected graphs, thus, we use the extension method in Section 8 in the following experiments.

9.1 Data sets

We prove the practicability of our approaches on the following datasets:

a) Erdos Renyi Model: This is a classical random graph model. It defines a random graph as N vertices connected by M edges, chosen randomly from the $N(N-1)/2$ possible edges. In experiments, we vary N from 10K to 100K. The default average degree is set to be 5. This dataset is denoted ER data.

b) Scale-Free Model: A scale-free network is a network whose degree distribution follows a power law distribution. It means that the fraction $P(k)$ of vertices having k neighbors in the network, where $P(k) \sim k^{-\gamma}$. Usually, $2 < \gamma < 3$ [1]. Thus, in our experiments, we set $\gamma = 2.5$. We use the graph generator $gengraphwin$ (www.cs.sunysb.edu/ algorith/ implement/viger/distrib/) to generate a scale-free network. This dataset is denoted SF data. Here, the numbers of vertices are also set to be from 10K to 100K.

In the above two datasets, the default number of vertex labels (denoted as $|L|$) is 250. We evaluate the numbers of vertex labels from 100 to 500 in Experiment 7.

c) HPRD (http://www.hprd.org/download) is a human protein interaction network consisting of 9,460 vertices and 37,000 edges. We used the GO term description as vertex labels. There are 307 vertex labels in HPRD.

d) Yago dataset (http://www.mpi-inf.mpg.de/yago-naga/yago/) is a RDF dataset. We build a RDF graph, in which vertices corresponds to subjects and objects, and edges correspond to properties. For each subject or object, we use its corresponding class as its vertex label. We ignore the edge labels in our experiments. There are 368,587 vertices, 543,815 edges and 45,450 vertex labels in Yago graph.

e) DBPedia dataset (http://dbpedia.org/About) is also a knowledge base. We build a resource graph, in which vertices corresponds to resources, and edges correspond to relations between two resources. For each resource in DBPedia, it is annotated by a Wikipedia

document. We use the article category of the document annotated the resource as its vertex label. We ignore the edge labels in our experiments. There are 1,117,572 vertices, 4,572,916 edges and 127,841 vertex labels in DBPedia graph.

Note that, in the GADDI algorithm, there are two important parameters, $Length$ (upper bound of the distance between a pair of indexed vertices) and k (radius) mentioned above. We set $Length$ and k to 2 and 4, respectively. The only parameter of SPath, i.e., the neighborhood scope k_0, is set 4.

9.2 Results

In this section, we do seven experiments to compare our methods to some competitors from different perspectives.

Experiment 1 (Performance VS. $|V(G)|$) This experiment is to study the scalability of our methods with increasing of $|V(G)|$. In this experiment, we use ER datasets and fix $|V(Q)|$ (i.e., the number of vertices in query Q) to be 10 and $|L|$ to be 250. Note that, TurboISO does not build up any indices, so TurboISO takes no time for the offline.

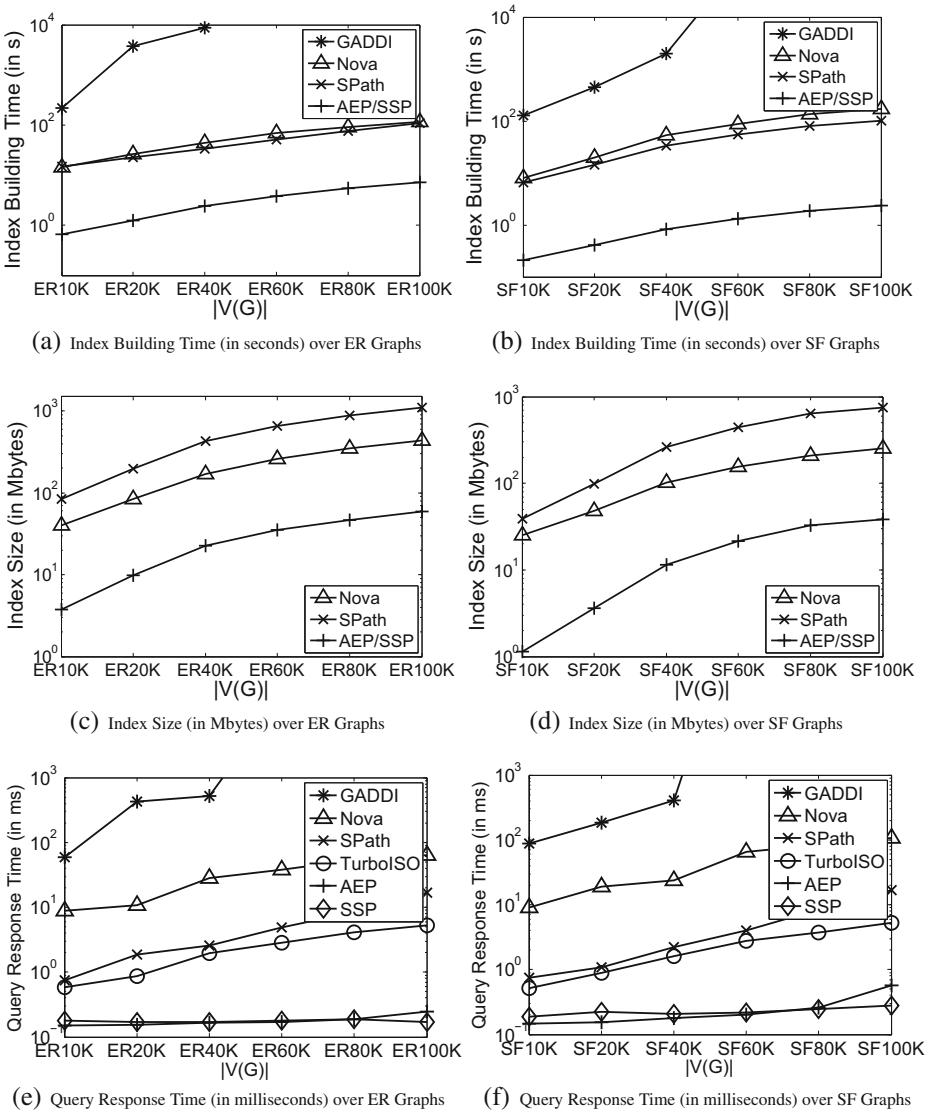
Figure 13a shows that our methods (AEP and SSP) have linear index building time, which outperforms Nova, GADDI and SPath by orders of magnitude. Figure 13c shows that our methods (AEP and SSP) have much smaller index sizes than those in Nova and SPath. Note that, GADDI cannot finish index building in reasonable time (within 24 hours) when $|V(G)| \geq 60K$. In addition, the GADDI software provided by authors cannot report index size, thus, we ignore the comparison with GADDI in index sizes. Here, AEP and SSP have the same index building process. Thus, they have the exact same index building time and index size, as shown in Figures 13a and c.

We also report the average query response times in Figure 13e over ER graphs, which show that both SSP and AEP do not increase greatly when varying $|V(G)|$ from 10K to 100K, which confirms the good scalability of our methods. Note that, our methods are faster than other methods by at least one order of magnitude in query processing. Furthermore, SSP is better than AEP in large data graphs, such as $|V(G)| = 100$K. The reason is that: SSP uses "star" instead of "edge" in AEP as building blocks in join processing, thus SSP has less intermediate results than that in AEP. When $|L|$ is fixed, there are more vertices having the same vertex label with the increasing of $|V(G)|$. Therefore, the performance differences of SSP and AEP are more clear when $|V(G)|$ is large. We also evaluate our method over SF graph in Figure 13.

Experiment 2 (Performance VS. Graph Degree) In this experiment, we use ER graphs, and fix $|V(G)|$ =100K and vary $d = |E(G)|/|V(G)|$ from 5 to 20. Since GADDI cannot work when $|V(G)|$ =100K, thus, we ignore the comparison with GADDI. Figure 13a shows that Nova cannot work when $d > 10$, and our method is much faster than Nova when $d = 5$ and 10.

More interestingly, with the increasing of d, the index building time is growing very slowly, as shown in Figure 14a. Figure 14b shows that our methods (AEP and SSP) are faster than Nova and TurboISO in query response time. Furthermore, SSP is better than AEP, especially when d increases to 20, which confirms the superiority of SSP method in dense graphs.

Experiment 3 (Performance versus $|E(Q)|$) In this experiment, we evaluate the performance of our methods with the increasing of query size, i.e.$|E(Q)|$. We first use ER graphs.



(a) Index Building Time (in seconds) over ER Graphs

(b) Index Building Time (in seconds) over SF Graphs

(c) Index Size (in Mbytes) over ER Graphs

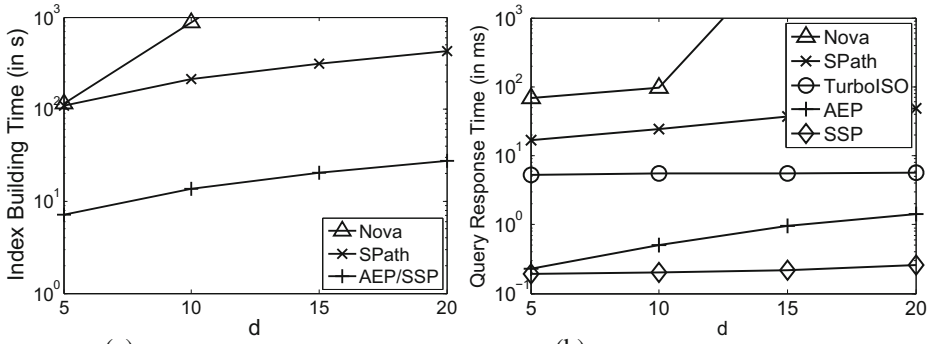
(d) Index Size (in Mbytes) over SF Graphs

(e) Query Response Time (in milliseconds) over ER Graphs

(f) Query Response Time (in milliseconds) over SF Graphs

Figure 13 Performance VS. $|V(G)|$

In this experiment, we first fix $|V(G)|$ to be 100K. Figure 15 shows that query response time are increasing in all methods when varying $|E(Q)|$ from 10 to 80. Our methods (AEP and SSP) have the best performance. Furthermore, SSP is much better than AEP in ER graphs, especially when $|E(Q)|$ is large. The reason is very clear: we need to perform more join steps when increasing $|E(Q)|$. However, the differences between AEP and SSP are not very clear in SF graphs. Furthermore, SSP is not as good as AEP in many queries in SF graphs, as shown in Figure 15b. The reason behind that is that a large fraction of vertices in SF graphs have very small degrees, which favor AEP algorithm. Actually, we find that SSP



(a) Index Building Time (in seconds) (b) Query Response Time (in milliseconds)

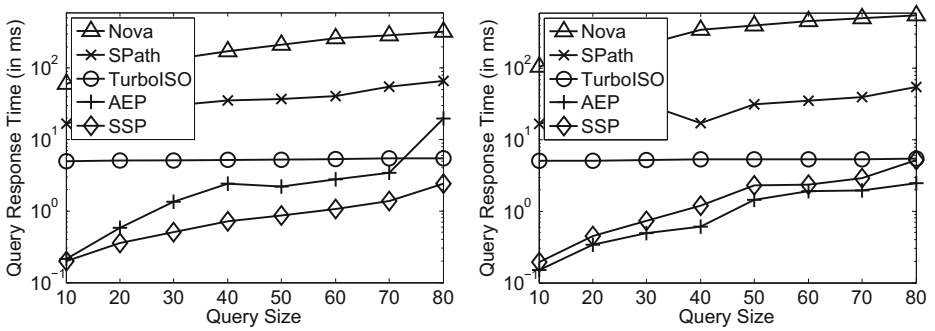
Figure 14 Performance VS. graph degree d in ER graphs

is much better than AEP in dense graphs, but the advantage is not clear in sparse graphs, as evaluated in Experiment 2.

Experiment 4 (Performance over Real Datasets) We also test our methods in two real datasets HPRD, Yago and DBPedia. Note that Nova and GADDI cannot work on Yago dataset due to running out of memory. Moreover, our own implementation of SPath can not work on DBPedia dataset. Therefore, we only compare our method with SPath and TurboISO in Yago dataset, and compare our method with TurboISO in DBPedia dataset.

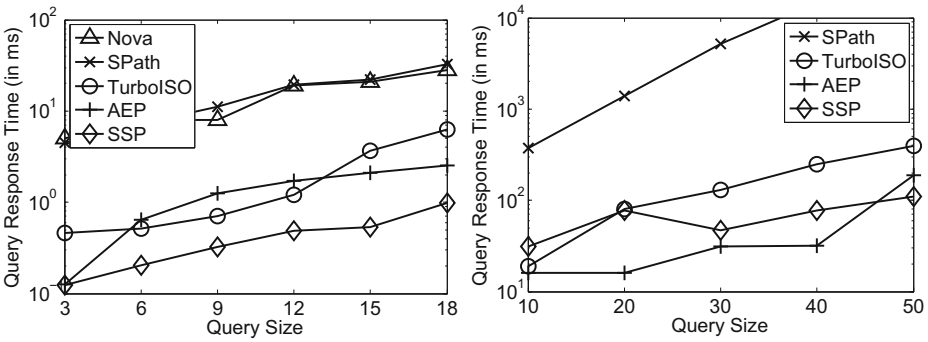
In HPRD dataset, out methods can finish the offline processing in less than 1 second, while Nova and GADDI need about 20 seconds and 600 seconds, respectively. The online processing in our methods are also faster than that in other methods, as shown in Figures 17a and b. In Yago dataset, we can finish index building in less than 3 minutes, which is much faster than that in SPath. Furthermore, SSP and AEP are both faster in query response time than that in SPath and TurboISO.

In DBPeida dataset, we can finish the offline processing in about 20 second. As shown in Figure 17c, when the query size is no larger than 6, both AEP and SSP are faster than TurboISO. When the query size is larger than 6, AEP becomes slower than TurboISO due to large number of join operations. However, SSP is still faster than TurboISO for the queries whose size is larger than 6.

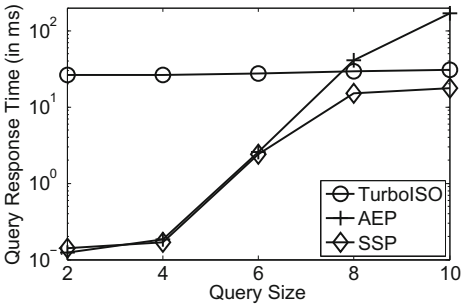


(a) Query Response Time (in milliseconds) over ER graphs (b) Query Response Time (in milliseconds) over SF graphs

Figure 15 Performance VS. $|E(Q)|$



(a) Query Response Time over HPRD (in milliseconds) (b) Query Response Time over Yago (in milliseconds)



(c) Query Response Time over DBPedia (in milliseconds)

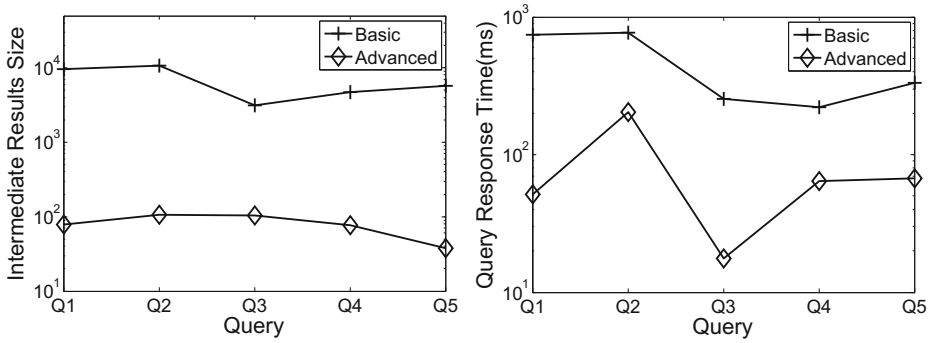
Figure 16 Online performance over real datasets

Experiment 5 (Reducing Intermediate Result Size) In this experiment, we show performance gains by reducing intermediate result size. We use five query graphs and fix $|V(Q)|$ to be 20. The data graph we use is HPRD. Figure 17a shows that our method reduces intermediate result size greatly, which means less join cost. Thus, the query performance (i.e. query response time) is improved significantly, as shown in Figure 17b.

Experiment 6 (Index Maintenance) As discussed early, our method can support online index maintenance efficiently. In this subsection, we evaluate the performance of our index maintenance method. Table 2 shows the average insertion time (for one edge) when varying the sizes of ER and SF graphs (i.e.$|V(G)|$) to be updated from 10K to 100K, respectively. For example, when $|V(G)| = 10K$, the average insertion time is 0.15 ms. An interesting finding is that the average insertion time in SF graph is larger than that in ER graphs, as shown in Table 2a. We also report the average insertion time over Yago dataset in Table 2b.

Table 2c shows that the average deletion time (for one edge) when varying the sizes of ER and SF graphs (i.e.$|V(G)|$) to be updated from 10K to 100K, respectively. Table 2c shows that ER and SF graphs have the similar deletion times. Table 2d reports the performance over Yago dataset.

Experiment 7 (Performance VS. $|L|$) Now, we study the performance of our methods with the increasing of $|L|$, i.e., the number of distinct vertex labels. In this experiment, we fix



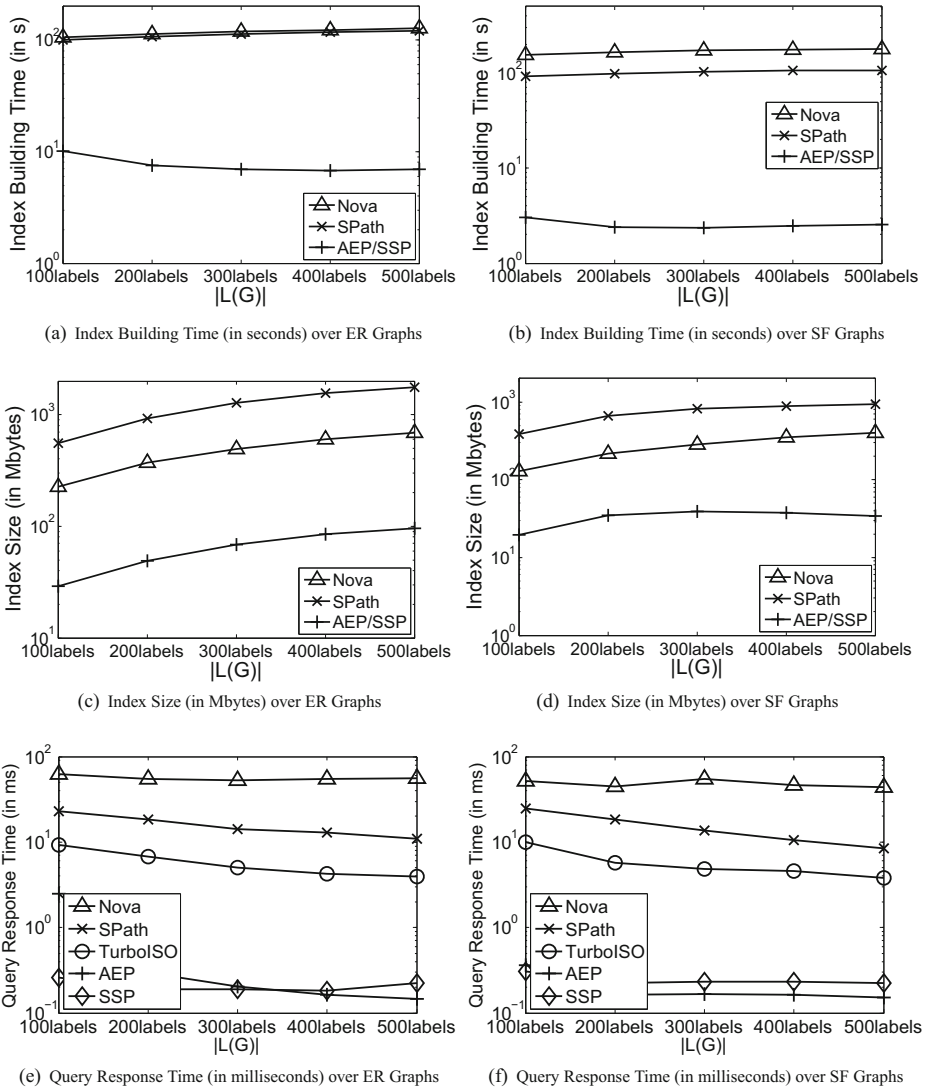
(a) Intermediate Results Size (b) Query Response Time (in milliseconds)

Figure 17 Performance improvement by reducing intermediate results

$|V(G)|$ to be 100K, and vary $|L|$ from 100 to 500 in this experiment. Since GADDI cannot run when $|V(G)| > 60K$, thus, we ignore the comparison with GADDI in this experiment. We first use ER graphs in our testing. Figure 18a shows that the index building time in our method is much faster than that in Nova and SPath. Also, we have smaller index sizes than others, as shown in Figure 18c. We also evaluate the online performance. Figure 18e shows that our method outperforms Nova, SPath and TurboISO by orders of magnitude. Furthermore, SSP is better than ASP when $|L|$ is small. When $|L|$ is small, there are more vertices having the same label. In this case, SSP has much smaller intermediate result size than that in AEP. We have the similar performance results in SF graphs, as shown in Figure 18b, d and f.

Table 2 Maintenance performance

(ms)	ER	SF	(ms)	Yago
(a) Insertion On Synthetic Data			(b) Insertion On Yago	
10K	0.15	0.235	100K	4.37
20K	0.15	0.315	200K	6.56
40K	0.15	0.39	300K	7.42
60K	0.16	0.47	400K	12.54
80K	0.16	0.625	500K	13.48
100K	0.16	0.625		
(c) Deletion On Synthetic Data			(d) Deletion On Yago	
10K	0.15	0.15	100K	5.1
20K	0.16	0.16	200K	7.19
40K	0.16	0.16	300K	11.72
60K	0.16	0.16	400K	11.77
80K	0.31	0.16	500K	15.47
100K	0.32	0.31		



(a) Index Building Time (in seconds) over ER Graphs

(b) Index Building Time (in seconds) over SF Graphs

(c) Index Size (in Mbytes) over ER Graphs

(d) Index Size (in Mbytes) over SF Graphs

(e) Query Response Time (in milliseconds) over ER Graphs

(f) Query Response Time (in milliseconds) over SF Graphs

Figure 18 Performance VS. $|L|$

9.3 Analysis

In this section, we discuss why our method can work better than existing approaches as our experiments show.

All traditional subgraph match algorithms [12, 37, 39, 42] adopt the vertex-oriented approach. Specifically, these methods build the indices to find candidates of each query vertex u_i, $i = 1, ..., |V(Q)|$, denoted as $C(u_i)$. The whole search space is $\prod_{i=1}^{|V(Q)|} |C(u_i)|$.

Different from existing approaches, our method employs the edge-oriented approach. For each query edge e_j in Q, $j = 1, ..., |E(Q)|$, the candidate edges are denoted as $C(e_j)$. Thus, the total join search space is $\prod_{j=1}^{|E(Q)|} |C(e_j)|$.

In the worst case, both our methods and previous works may fail to prune any candidates. Therefore, the candidates' number for a query vertex is equal to the number of vertices with the same label in data graph. Similarly, the candidates' number of each query edge is equal to the number of edges with the same label in data graph.

We suppose that all the labels are distributed evenly on vertices in data graph. Thus, the average candidates' number of each query vertex is $\frac{|V(G)|}{|L|}$. On the contrary, the average candidates' number of each query edge is $\frac{|E(G)|}{|L|^2}$. Hence, the average search spaces of the previous solutions and our approaches are $(\frac{|V(G)|}{|L|})^{|V(Q)|}$ and $(\frac{|E(G)|}{|L|^2})^{|E(Q)|}$.

In real datasets, for one thing, the graphs are very sparse, whether data graph or query graph. Usually, if the degree of a real graph is larger than 10, we can say that this graph is dense in real use. For instance, the degrees of HPRD and Yago are about 3.9 and 1.48, respectively. For another, the number of labels is much larger than the degree of a graph. A real graph often has hundreds of labels or more. For example, the labels' numbers of HPRD and Yago are about 307 and 45,450, which is much larger than their degrees.

Therefore, we can know that $\frac{|E(G)|}{|V(G)|} \ll |L|$. Then, we can know that $\frac{|V(G)|}{|L|} \ll \frac{|E(G)|}{|L|^2}$.

Moreover, in practice, the size and degree of query graph are very small, so $|E(Q)|$ is approximately equal to $|V(Q)|$. Hence, in real application, $(\frac{|V(G)|}{|L|})^{|V(Q)|}$ is larger than $(\frac{|E(G)|}{|L|^2})^{|E(Q)|}$.

As a result, our methods is more efficient than the previous methods as our experiments show.

10 Conclusions

In order to address subgraph query over a single large data graph G, in this paper, we first map G into two-dimensional space $\Re^2$, and then propose a novel bitmap structure to index $\Re^2$. At run time, we propose two kinds of subgraph query algorithm in this paper. Aimed by the bitmap index, we can reduce the search space and improve the query performance significantly. Extensive experiments over both real and synthetic data sets confirm that our methods outperforms existing ones in both offline and online performances by orders of magnitude.

References

1. Albert, R., Barabási, A.-L.: Statistical mechanics of complex networks. Rev. Mod. Phys., 74 (2002)
2. Chan, E.P.F., Lim, H.: Optimization and evaluation of shortest path queries. VLDB J. **16**(3) (2007)
3. Chen, Y., Chen, Y.: An efficient algorithm for answering graph reachability queries. In: ICDE (2008)
4. Cheng, J., Ke, Y., Ng, W., Lu, A.: fg-index: Towards verification-free query processing on graph databases. In: SIGMOD (2007)
5. Cheng, J., Yu, J.X.: On-line exact shortest distance query processing. In: EDBT (2009)
6. Cohen, E., Halperin, E., Kaplan, H., Zwick, U.: Reachability and distance queries via 2-hop labels. SIAM J. Comput. **32**(5) (2003)
7. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: A (sub)graph isomorphism algorithm for matching large graphs. IEEE Trans. Pattern Anal. Mach. Intell. **26**(10), 1367–1372 (2004)

8. Fan, W., Li, J., Ma, S., Tang, N., Wu, Y., Wu, Y.: Graph pattern matching: From intractable to polynomial time. PVLDB **3**(1), 264–275 (2010)
9. Fan, W., Li, J., Wang, X., Wu, Y.: Query preserving graph compression. In: SIGMOD Conference, pp. 157–168 (2012)
10. Fan, W., Wang, X., Wu, Y.: Answering graph pattern queries using views. In: IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014, pp. 184–195 (2014)
11. Gao, J., Zhou, C., Zhou, J., Yu, J.X.: Continuous pattern detection over billion-edge graph using distributed framework. In: IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014, pp. 556–567 (2014)
12. Han, W.-S., Lee, J., Lee, J.-H.: Turboiso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In: SIGMOD Conference, pp. 337–348 (2013)
13. He, H., Singh, A.K.: Closure-tree: An index structure for graph queries. In: ICDE (2006)
14. He, H., Singh, A.K.: Graphs-at-a-time: query language and access methods for graph databases. In: SIGMOD Conference, pp. 405–418 (2008)
15. http://giraph.apache.org
16. Jiang, P.Y.H., Wang, H., Zhou, S.: Gstring: A novel approach for efficient search in graph databases. In: ICDE (2007)
17. Jing, N., Huang, Y.-W., Rundensteiner, E.A.: Hierarchical encoded path views for path query processing: An optimal model and its performance evaluation. IEEE Trans. Knowl. Data Eng. **10**(3) (1998)
18. Khan, A., Li, N., Yan, X., Guan, Z., Chakraborty, S., Tao, S.: Neighborhood based fast graph search in large networks. In: SIGMOD Conference, pp. 901–912 (2011)
19. Klein, K., Kriege, N., Mutzel, P.: Ct-index: Fingerprint-based graph indexing combining cycles and trees. In: ICDE, pp. 1115–1126 (2011)
20. Lee, J., Han, W.-S., Kasperovics, R., Lee, J.-H.: An in-depth comparison of subgraph isomorphism algorithms in graph databases. PVLDB **6**(2), 133–144 (2012)
21. Murty, K.G.: Network Programming. Prentice Hall (1992)
22. Sakr, S., Elnikety, S., He, Y.: G-sparql: a hybrid engine for querying large attributed graphs. In: CIKM, pp. 335–344 (2012)
23. Shao, B., Wang, H., Li, Y.: Trinity: a distributed graph engine on a memory cloud. In: SIGMOD Conference, pp. 505–516 (2013)
24. Shasha, D., Wang, J.T.-L., Giugno, R.: Algorithmics and applications of tree and graph searching. In: PODS (2002)
25. Sun, Z., Wang, H., Wang, H., Shao, B., Li, J.: Efficient subgraph matching on billion node graphs. PVLDB **5**(9), 788–799 (2012)
26. Tian, Y., McEachin, R.C., Santos, C., States, D.J., Patel, J.M.: Saga: a subgraph matching tool for biological graphs. Bioinformatics **23**(2) (2007)
27. Tian, Y., Patel, J.M.: Tale: A tool for approximate large graph matching. In: ICDE, pp. 963–972 (2008)
28. Tong, H., Faloutsos, C., Gallagher, B., Eliassi-Rad, T.: Fast best-effort pattern matching in large attributed graphs. In: KDD, pp. 737–746 (2007)
29. TriSSl, S., Leser, U.: Fast and practical indexing and querying of very large graphs. In: SIGMOD (2007)
30. Ullmann, J.R.: An algorithm for subgraph isomorphism. J. ACM **23**(1) (1976)
31. Wang, H., He, H., Yang, J., Yu, P.S., Yu, J.X.: Dual labeling: Answering graph reachability queries in constant time. In: ICDE (2006)
32. Williams, J.H.D.W., Wang, W.: Graph database indexing using structured graph decomposition. In: ICDE (2007)
33. Yan, X., Yu, P.S., Han, J.: Graph indexing: A frequent structure-based approach. In: SIGMOD (2004)
34. Yang, S., Wu, Y., Sun, H., Yan, X.: Schemaless and structureless graph querying. PVLDB **7**(7), 565–576 (2014)
35. Zeng, Z., Tung, A.K.H., Wang, J., Feng, J., Zhou, L.: Comparing stars: On approximating graph edit distance. PVLDB **2**(1), 25–36 (2009)
36. Zhang, S., Hu, M., Yang, J.: Treepi: A novel graph indexing method. In: ICDE (2007)
37. Zhang, S., Li, S., Yang, J.: Gaddi: distance index based subgraph matching in biological networks. In: EDBT, pp. 192–203 (2009)
38. Zhang, S., Yang, J., Jin, W.: Sapper: Subgraph indexing and approximate matching in large graphs. PVLDB **3**(1), 1185–1194 (2010)
39. Zhao, P., Han, J.: On graph query optimization in large networks. In: VLDB (2010)
40. Zhao, P., Yu, J.X., Yu, P.S.: Graph indexing: Tree + delta >= graph. In: VLDB (2007)
41. Zheng, W., Zou, L., Lian, X., Wang, D., Zhao, D.: Graph similarity search with edit distance constraint in large graph databases. In: CIKM, pp. 1595–1600 (2013)

42. Zhu, K., Zhang, Y., Lin, X., Zhu, G., Wang, W.: Nova: A novel and efficient framework for finding subgraph isomorphism mappings in large graphs. In: DASFAA (1) (2010)
43. Zou, L., Chen, L., Özsu, M.T.: Distancejoin: Pattern match query in a large graph database. PVLDB **2**(1), 886–897 (2009)
44. Zou, L., Chen, L., Yu, J.X., Lu, Y.: A novel spectral coding in a large graph database. In: EDBT (2008)