Continuous Subgraph Pattern Search over Graph Streams

Changliang Wang and Lei Chen

Department of Computer Science and Engineering
Hong Kong University of Science and Technology
{sonicwcl, leichen}@cse.ust.hk

***Abstract*— Search over graph databases has attracted much attention recently due to its usefulness in many fields, such as the analysis of chemical compounds, intrusion detection in network traffic data, and pattern matching over users' visiting logs. However, most of the existing work focuses on search over static graph databases while in many real applications graphs are changing over time.**

In this paper we investigate a new problem on continuous subgraph pattern search under the situation where multiple target graphs are constantly changing in a stream style, namely the subgraph pattern search over graph streams. Obviously the proposed problem is a continuous join between query patterns and graph streams where the join predicate is the existence of subgraph isomorphism. Due to the NP-completeness of subgraph isomorphism checking, to achieve the real time monitoring of the existence of certain subgraph patterns, we would like to avoid using subgraph isomorphism verification to find the exact query-stream subgraph isomorphic pairs but to offer an approximate answer that could report all probable pairs without missing any of the actual answer pairs. In this paper we propose a light-weight yet effective feature structure called Node-Neighbor Tree to filter false candidate query-stream pairs. To reduce the computational cost, we further project the feature structures into a numerical vector space and conduct dominant relationship checking in the projected space. We propose two methods to efficiently check dominant relationships and substantiate our methods with extensive experiments.

I. Introduction

As one of the most popular data models, graph has been used in various real applications such as social network modeling and chemical compound analysis. Due to their wide usages, many interesting graph problems are extensively studied, for example, graph reachability [22], [21], subgraph search [24], [17], [4], and keyword search in graphs [10], [7].

In fact, in many applications, graphs are often evolving along the time in a stream fashion instead of remaining static. For example, in a traffic network, the links between nodes are changing over the time. Given another example, during a chemical reaction, the structures of chemical compounds often change along the reaction process. We can model these evolving graphs as *graph streams*, i.e., a sequence of graphs which grows indefinitely over time [18]. However, most of the previous work assumes that graph data are rather static, which raises challenges when applying to graph streams. Compared to static graphs, graph streams not only inherit the complexity of graphs but also possess their own characteristics: 1) graphs are frequently updated, and 2) real time response is necessary.

In this paper, we study the problem of *continuous subgraph pattern search* over graph streams. *Subgraph search* has been used as an effective tool for finding useful substructures in a graph database. For example, a bio-chemist can utilize the subgraph search to analyze the functionality of newly found chemical compounds; network security administrators can conduct a pattern (subgraph) matching over the network traffic data to detect possible malicious attacks. Formally, subgraph search over a graph database $\mathcal{D}$ is defined as follows:

Given a query graph Q, we need to find all data graphs $G_i \in \mathcal{D}$, where G_i contains the query Q, namely, Q is subgraph isomorphic to G_i.

Due to the NP-completeness of subgraph isomorphism checking [5], most of the previous works on subgraph search employ a filter-and-verify strategy to reduce the number of isomorphism checking. Specifically, graphs in the database are indexed by a set of distinguishing features, such as paths [17], trees [28] and subgraphs [24], then during query processing, the extracted features are first used to prune the graphs that do not contain the query graph, and afterward the left candidate graphs are verified by the subgraph isomorphism checking.

Unfortunately, we can not directly apply the previous methods for subgraph search over static graphs to graph streams due to their unique characteristics. For example, gIndex [24] needs to mine frequent subgraphs(features) at each timestamp, which does not satisfy the real time response requirement of graph streams. Given another example, GraphGrep [17] may satisfy the real time response requirement, however, it only uses paths to filter out candidates and many false positives (i.e. not the actual results that are reported as positive) still exist in the result after filtering.

Motivated by shortcomings of previous approaches and the challenges raised by graph streams, we address the problem of continuous subgraph search over graph streams, that is, given a set of predefined query graphs (patterns), we *continuously* monitor a set of graph streams and report the *possible* appearances of a set of subgraphs (patterns) in a set of graph streams at each timestamp (The formal problem definition is given in Section II). In this work, in order to satisfy the real time response requirement of search over graph streams, we focus on retrieving the *possible* appearance instead of exact appearance of the subgraph. Here possible appearance

1084-4627/09 $25.00 © 2009 IEEE
DOI 10.1109/ICDE.2009.132

means the *pattern-graph* matching candidate pairs are detected using an approximate method without conducting the subgraph isomorphism checking. Furthermore, we also require that the proposed approximate method should not introduce *false negatives* (i.e. the actual answers that are missed) and has as less *false positives* as possible.

Obviously, the general problem that we target on is a continuous "join" operation between query patterns and graph streams under the predicate "subgraph isomorphism". One real application of this problem is to detect malicious attacks over a set of traffic stream data where the possible malicious attacks that are derived from some domain knowledge are modeled as a set of traffic graph patterns. We need to report all the possible attacks in real time, i.e., to find the possible matches between patterns and traffic data whenever the traffic data are updated. Any delay or missing of the real attacks will cause millions of dollars' loss.

Based on the requirements of no false negatives and real time response, we reinvestigate the previously proposed filtering methods for static graph databases and propose a light-weight yet effective structure, *Node-Neighbor Tree*(NNT), as a filtering feature. Compared to paths, Node-neighbor trees capture more structural information, and thus have more pruning power (i.e. resulting less false positives). Moreover, generating node-neighbor trees does not require performing any mining process. However, although using subtree as filtering feature is more effective than only using paths, subtree isomorphism verification is still expensive. To further reduce this filtering cost, we propose a novel encoding method to encode each node neighbor tree into a numerical vector. With such a conversion, we can search for the possible pattern-graph matching pairs by only checking the *dominant relationship* between two set of converted numerical vectors from query patterns and graph streams, respectively. To speed up the search process in the converted vector space, we propose two efficient methods to check dominant relationship.

To summarize, we list our contributions as follows,

- We propose a new problem, continuous subgraph search over graph streams and design an effective and light-weight feature structure, node neighbor tree, to reduce false negatives and computational cost;
- We further improve the filtering efficiency by introducing a novel encoding method to transform node-neighbor trees into numerical vectors and replacing the expensive subtree isomorphism verification with a dominant relationship checking in the converted space;
- We propose two different strategies to verify the dominant relationship between two vector sets efficiently;
- Finally, through extensive experiments, we demonstrate the effectiveness and efficiency of our proposed methods in finding approximate matching pairs between query patterns and graph streams.

The rest of the paper is arranged as follows, we formally define the problem of continuous subgraph search over graph streams and related concepts in Section II. In Section III, a new graph feature, node-neighbor tree is presented for pruning false positives during the search. We present the novel encoding method and the dominant relationship checking problem in Section IV. Extensive experiments on real and synthetic graph stream data have demonstrated the effectiveness as well as efficiency of our proposed methods in Section V. We discuss and compare with related work in Section VI. Finally, we conclude in Section VII.

II. Problem Definition

A. Graphs and Graph Streams

Definition 2.1: **Graph.** A labeled graph is denoted as $G =< V(G), E(G), label_v, label_e >$, where $V(G)$ is the set of vertices,$E(G)$ is the set of edges, and $label_v$ and $label_e$ are the functions to assign labels to $V(G)$ and $E(G)$ respectively.

Definition 2.2: **Subgraph.** Given a graph G', if its vertices and edges form subsets of that of a given graph G, and G' has the same labeling functions as G, G' is a subgraph of G.

Definition 2.3: **Subgraph Isomorphism.** A subgraph isomorphism is an injective function f: $V(G) \rightarrow V(G')$, such that (1) $\forall\ u \in V(G)$, $label_v(u) = label'_v(f(u))$, (2) $\forall\ (u,v) \in E(G)$, $(f(u), f(v)) \in E(G')$ and $label_e(u,v) = label'_e(f(u), f(v))$, where $label()$ and $label'()$ are the label functions of G and G', respectively.

Definition 2.4: **Graph Change Operation** Given a graph G, an edge (u,v) insertion/deletion in G can be represented as a triple, $< op, u, v >$, where $op = ins/del$ means the edge insertion/deletion, and u, v are the vertices of G. Insertion of a new node w can be represented by a set of edge insertions which insert all the newly created edges between the vertices in G and w, similarly, the deletion of a node w can be considered as a set of edge deletions which delete all the edges associated with the deleting vertex w. Therefore, graph change operations refer to a set of edge insertions or deletions. Given a graph change operation, $GC = \{< op_1, u, v >, \ldots, < op_k, u, v >\}$ $(k \geq 1)$, G can be changed to G' by applying GC to G denoted as $GC \rightarrow G = G'$.

Note that in Definition 2.4, we do not consider the case of inserting an isolated vertex. In other words, we assume graph at each timestamp is connected, which is also assumed in many works [28], [24], [26]. In the stream scenario, it is quite common to assume that changes between consecutive timestamps are not much, which is also referred as temporal locality in stream data processing [13], [6]. Temporal locality assumption is valid as well for real applications over graphs. For example, in a social network, one person's friendship links are growing locally and smoothly. Based on this assumption, we can infer that graph change operations are not many between two consecutive graphs. Given a sequence of graph change operations, we define graph change stream as follows:

Definition 2.5: **Graph Change Operation Stream.** Given a starting graph G, a graph change operation stream ΔGC is a sequence of graph change operations GC_t, with t denoting the timestamp. ΔGC can be denoted as:

$$\Delta GC := [GC_1, GC_2, ..., GC_t, ...]$$

Now, we define *graph streams* based on the graph change operation streams:

Definition 2.6: **Graph Stream.** *Given a starting graph G, and a graph change stream ΔGC, the graph stream GS, is the sequence of the corresponding graphs defined as follows.*

$$GS = \{G_0, G_1, G_2, G_t ...\},$$
$$G_0 = G \qquad G_t = GC_t \rightarrow G_{t-1}$$

Figure 1 shows an example of graph stream. The left most graph is the starting graph, then from left to right, there are three graph change operations for timestamp 1 to 3 respectively. The right most graph is the graph at timestamp 3, which is exactly $GC_3 \rightarrow (GC_2 \rightarrow (GC_1 \rightarrow G_1)))$.

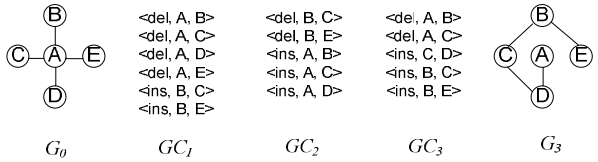


Fig. 1. Illustration of a graph change operation stream

In the rest of paper, we use term "graph streams" referring a sequence of graphs (GS) and "stream graph" referring one of the graphs (G_i) in the sequence of graphs.

B. Problem formulation

In this paper, we focus on answering *continuous subgraph patterns* over graph streams. More specifically, we assume a user has a set of subgraph patterns and starts monitoring graph streams from timestamp 0, then as time evolves, the user wants the system *continuously report* the appearances of certain subgraph patterns on the graph streams at each timestamp. Formally, we define the problem as follows:

Definition 2.7: **Subgraph Search Over Graph Streams.** Given a set of graph streams $< GS_1, GS_2, \ldots, GS_{k_1} >$ and a set of query graphs $< Q_1, Q_2, \ldots, Q_{k_2} >$, we want to continuously report all the *joinable* pairs $< GS_{(i,t)} Q_j >$ at each timestamp t, where $joinable$ means Q_j is a subgraph of $GS_{(i,t)}$, $1 \leq i \leq k_1$, $1 \leq j \leq k_2$, and $t >= 0$.

Note that in Definition 2.7, similar to [6], [13], we assume the set of query pattern graphs derived from some domain knowledge in a monitoring application is fixed. We leave the dynamic set of query patterns as an interesting future work. According to Definition 2.7, if we want to find the $joinable$ pairs, we have to conduct subgraph isomorphism checking to ensure the appearances of certain patterns on graph streams. However, since subgraph isomorphism checking is an NP-complete problem [5], it is not possible for us to satisfy the real time response requirement of graph stream processing, such as operations on the network traffic streams. Therefore, we focus on a little bit relaxed problem, *approximate subgraph queries over graph streams*, which is defined as follows:

Definition 2.8: **Approximate subgraph Search Over Graph Streams.** Given a set of graph streams $< GS_1, GS_2, \ldots, GS_{k_1} >$ and a set of query graph patterns $< Q_1, Q_2, \ldots, Q_{k_2} >$, we continuously report all *possible joinable* pairs $< GS_{(i,t)} Q_j >$ at each timestamp t, where *possible joinable* means Q_j is probably a subgraph of $GS_{(i,t)}$, $1 \leq i \leq k_1$, $1 \leq j \leq k_2$, and $t >= 0$.

For an approximate subgraph search, we do not require reporting the exact appearances of patterns on graph streams. Instead, all the possible appearances of the subgraph patterns should be promptly detected once data graphs enter the system. In addition, the solution should not introduce any false negatives and report few false positives.

Algorithm	Average Time	Candidate size
Our method	1240ms	16.3%
gIndex	7210ms	13.3%
GraphGrep	5480ms	67.6%

Fig. 2. Efficiency and Effectiveness comparison

III. A Novel Feature Structure for Filtering

To address the problem defined in Definition 2.8, we can employ previously proposed filtering methods, such as gIndex [24] and GraphGrep [17], to report the possible joinable pairs. However, most of the existing work [24], [28] rely on frequent subgraph (sub-tree) mining algorithms to extract features for filtering, which makes the methods inefficient in detecting patterns over graph streams. Different from gIndex, GraphGrep enumerates all pathes with lengthes up to a given value L to filter out false positives. We can directly apply GraphGrep in graph stream setting, however, it is time consuming if we set L to be a larger value, i.e. 10, for good effectiveness. On the other hand, setting L to a smaller value, i.e. 4, although efficient, will introduce many false positives.

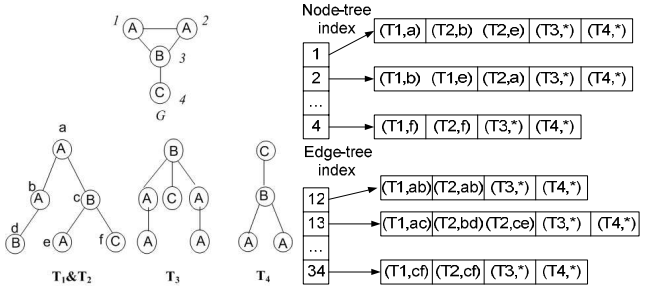
Figure 2 lists a preliminary test result of the three algorithms (gIndex, Graphgrep, and our proposed method) with respect to the average query processing time and the average candidate size per timestamp. In this test, we use 70 subgraph patterns and 70 graph streams. The processing time is measured by milliseconds. Candidate size is computed as the ratio between the number of possible joinable pairs returned and the total number of query-stream pairs. From the results, we can observe that gIndex has the minimal candidate size but consumes much more time than the other two algorithms do. Though GraphGrep requires less time, it reports more than half of the total pairs as candidates, which makes the result less useful.

A. Node-neighbor Tree

In this section, we present a novel feature, node-neighbor tree (NNT), which captures the local structure around each vertex. Most importantly, a node-neighbor tree does not mine features and has much higher pruning power than GraphGrep. The formal definition of NNT is as follows:

Definition 3.1: **Node-neighbor tree** Given a graph G and a depth value l, for any vertex $u \in G$, the NNT of u, denoted as $NNT(u)$, is a tree rooted at u and contains all the simple paths up to length l from u in G. A simple path refers to a path without repeated edges.

An example graph G together with NNTs of all its vertices and edges under $l = 2$ is shown in Figure 3(a). In the example,



(a) A Graph and NNTs (b) Node and Edge tree indexes

Fig. 3. Graph, Node-neighbor trees, and Index

G has four vertexes with ids from 1 to 4, which have labels A,A,B,C respectively. The NNTs of vertices 1 and 2 have the same structure, thus, we use only one tree to represent T_1 and T_2 in the example. T_3 rooted at vertex 3 has two branches consisting of the same labels A, which indicates that node 3 has two distinct neighbors with label A. T_4 has two different branches rooted at a node with label B.

For node-neighbor trees, we have the following lemma:

Lemma 3.1: If query graph Q is a subgraph of a data graph G, for each vertex $u \in Q$, there must exist a vertex $v \in G$ satisfying u's $NNT(u)$ is a subtree of v's $NNT(v)$.

Lemma 3.1 provides a necessary condition to filter out query graphs that are not contained in the stream graphs. Specifically, for each vertex u in the query graph, we check whether there is a node v in the stream graph satisfying u's node-neighbor tree is contained in v's. Totally, we need to conduct $O(n_1 * n_2)$ comparisons if the query graph has n_1 nodes and the stream graph has n_2 nodes. For each comparison, we need to perform subtree isomorphism checking, of which the complexity is $O((|T_1|^{1.5}/\log|T_1|)|T_2|)$ [15], where $|T_1|$ and $|T_2|$ are the sizes of two comparing trees. However, this computational cost of NNT filtering is not cheap based on the above analysis. Later, in Section IV, we present a novel encoding method to reduce the filtering cost significantly.

In addition to reducing the filtering cost, there is another key issue has to be solved before we can apply NNT over graph streams, that is how to incrementally maintain the NNT tree sets when a new data graph comes. To resolve this issue, we create a node tree index to show the appearances (positions) of nodes in each node-neighbor tree generated from a graph G. The position of each node in a NNT is labeled by an unique id within the node set of that NNT. In Figure 3 (a), the label besides each tree node in T_1 (T_2) is the position label of that node. Figure 3(b) shows an example of node-tree index for NNTs in Figure 3 (a). In the example index, node 2 appears in position b and e of tree T_1 rooted at node 1, thus in the entry 2 of the node-tree index, appearances (T_1, b) and (T_1, e) are stored. We also build an edge-tree index to denote the appearances of edges in each node-neighbor tree. Again, as shown in the Figure 3 (b), edge 13 appears as edge (a, c) in T_1 and as edges (b, d) and (c, e) in T_2. Thus in the entry 13 of the edge-tree index, (T_1, ab), (T_2, bd), and (T_2, ce) are stored.

When a new graph change operation arrives, we update the tree sets accordingly. We sequentialize a graph change operation into a series of insertions and deletions of edges and process all deletions first and then insertions. For deleting an edge, we locate all appearances of the edge and delete their subtrees and update all related entries. For inserting an edge, we locate all appearances of the nodes associated with the edge and append subtrees and update all related entries. The deletion steps are illustrated in Procedure Delete_Edge (Figure 4). To delete an edge e, we first retrieve all the appearances of e from the edge-tree index (line 1 in 4). Each appearance (T_i, u, v) means in node i's NNT, nodes u and v correspond to the two nodes of e in the graph. Line 3-7 uses a stack to traverse the subtree under edge (u, v) in a DFS style, and delete nodes and edges along this traversal. When an edge x in the subtree is visited (line 5), we delete its entry in I_{et} and two associated nodes' entries in I_{nt}(line 6). If x has children edges, all will be pushed into stack (line 7). Then the procedure will choose an edge at the top of the stack to visit, unless the stack is empty (line 4), which means the subtree is deleted completely thus the deletion for current appearance is finished. Take figure 3 for example. Suppose we are going to delete edge (1,3), we first find an appearance as (a, c) in $T1$. After executing line 3 to 7, (a, c), (c, e), (c, f) are all deleted from $T1$ and indexes. Then we continue to find an appearance (b, d) in $T2$, and then an appearance (c, e) in $T2$. This process goes on until all appearances of (1,3) are deleted.

Procedure Insert_Edge (Figure 5) lists the detailed steps to insert an edge. To insert an edge $e(a, b)$, we first retrieve all the appearances of node a from the node-tree index(line 1 in 5). Each appearance (T_i, u) means in node i's NNT, node u corresponds to a in the graph. We first append an edge (u, b) to u and update e's entry in I_{et} and b's entry in I_{nt}(line 3). To append the subtree under (u, b), we adopt a BFS style expansion(line 4-9). Each time we pop up the head of the queue(line 6) and check if current node can be expanded further(line 7). If yes, we enumerate all v's associated edges in the graph(line 8). If edge (v, x) does not appear in the path from root to node v, it means the path expanding from v to x will remain a simple path. Thus x is pushed to the back of the queue, edge (v, x) is added to I_{et} and x is added to I_{nt}(line 9). After updating a's appearances, we update b's appearances using the same procedure (line 10-11). Assume that we want to insert edge (1,4) to G, we first find 1's appearance at a in $T1$. We add branch $(A- > C- > B)$ to a. Then we find appearance at b in $T2$. We add branch $(A- > C)$ to b. This process continues until we expand all appearances of node 1. Then we apply the similar processing to node 4.

Lemma 3.2: The over all complexity for deleting (inserting) an edge to NNTs of graph G is $O(r^{l-1})$, where r is the maximum node degree of G.

.

IV. JOINING STREAMS WITH QUERIES

As discussed in the previous section, though NNTs have greater pruning power than paths, we have to conduct subtree

Procedure Delete_Edge {
Input: Edge e, edge-tree index I_{et}, node-tree index I_{nt}
Output: Updated indexes
(1) retrieve appearances of e in I_{et}
(2) for each appearance (T_i, u, v)
(3) push (u, v) to stack st
(4) While st not empty do
(5) $x \leftarrow st$.pop()
(6) delete x's entry in I_{et} and I_{nt}
(7) push children edges of x to st
(8) return I_{et} and I_{nt}
}

Fig. 4. Delete an edge

Procedure Insert_Edge {
Input: Edge (a, b), edge-tree index I_{et}, node-tree index I_{nt}
Output: Updated indexes
(1) retrieve appearances of a in I_{nt}
(2) for each appearance in (T_i, u)
(3) append an edge (u, b) at node u; update I_{et},I_{nt}
(4) put b in queue qu
(5) while qu not empty
(6) $v \leftarrow qu$.pop_head()
(7) if v's depth in tree=max depth then goto(5)
(8) for each edge (v, x) associated with v in graph
(9) if path from root of T_i to v does not contain edge (v, x) then create a node x and set it as v's child and put it to qu and update I_{et} and I_{nt}
(10) retrieve appearances of b in I_{nt}
(11) conduct the similar processing as a's
(12) return I_{et} and I_{nt}
}

Fig. 5. Insert an edge

isomorphism checking between the NNTs of a pattern graph and a stream graph, which is not efficient enough. In this section, we propose a novel encoding method to transform a NNT to a set of vectors and approximate subtree isomorphism checking by dominant relationship verification between two vector sets. Most importantly, the proposed encoding method will not introduce false negatives. In following discussion, for simplicity, all examples are illustrated with a scenario of join between single stream and single query pattern.

A. Projecting to numerical vectors

The inefficiency of NNT origins from subtree isomorphism checking. In fact, if a tree T_1 is a subtree of another tree T_2, all the simple pathes (branches) of T_1 should be included in the branches of T_2. Thus we can check if all of the simple pathes of T_1 are contained in T_2. Once we find there exists one path in T_1 that is not contained in T_2, we can conclude that T_1 is not a subtree of T_2. So instead of verifying subtree isomorphism, we approximate it by comparing the simple paths of two NNTs. We have the following lemma.

Lemma 4.1: Given a query graph Q and a stream graph G, if Q is subgraph isomorphic to G , for any vertex $u \in Q$, there must exist a vertex $v \in G$ satisfying that $NNT(u)$ is branch compatible to $NNT(v)$, where branch compatible means that all the simple paths (branches) of $NNT(u)$ are contained in the branches of $NNT(v)$.

Lemma 4.1 provides a $O(n_1 n_2 r^l)$ filtering method, which is much lighter than subtree isomorphism checking ($O(n_1 * n_2 * (|T_1|^{1.5}/\log|T_1|)|T_2|)$), where n_1 and n_2 are the number of nodes in stream and query graphs, and $|T_1|$, $|T_2|$ are the maximum sizes of NNTs in two graphs. However, for verifying whether Q is a subgraph of G, we need to check branch compatibility between every possible pair of nodes. In order to speed up this process, we propose a projection scheme to map a node-neighbor tree into a numerical vector, called *node-projected vector* (NPV). Before giving the definition of NPV, we first define the *dimension* of NPV as follows:

Definition 4.1: **Dimension** Given a node-neighbor tree, T, a dimension of the projected vector from T is defined as a triple $< l, lab_1, lab_2 >$, where lab_1, lab_2 are the corresponding node labels for a distinct edge E located in the depth l of T.

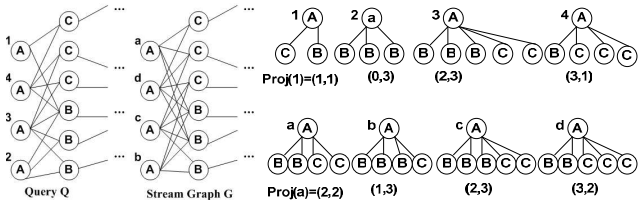
Figure 7 (a) shows an example of dimension derived from a query graph Q (left) and a stream graph G (right). For the purpose of demonstration, we only show the NNTs of vertices $\{1, 2, 3, 4, a, b, c, d\}$ with $l = 1$. Thus, there are only two dimensions from the NNTs, $(1, A, C)$, and $(1, A, B)$. Based on these two dimensions, we can apply Procedure Tree_Projection in Figure 6 to project a NNT into a *node projected vector* (NPV), which is defined as follows.

Definition 4.2: **Node Projected Vector** Given a vertex v in a graph G, and n dimensions, Dim_1, ..., Dim_i, Dim_n, (defined in Definition 4.1), the *node projected vector* of vertex u ($NPV(u)$) is a vector storing the numbers of appearances in each Dim_i in $NNT(u)$.

As shown in Figure 7(b), based on the two dimensions, $(1, A, C)$, and $(1, A, B)$. The NPVs of nodes $\{1, 2, 3, 4\}$ in Q and nodes $\{a, b, c, d\}$ in G are $\{(1, 1), (0, 3), (2, 3), (3, 1)\}$ and $\{(2, 2), (1, 3), (2, 3), (3, 2)\}$, respectively.

Procedure Tree_Projection {
Input: A NNT tree up to depth of l
Output: Node projected set
(1) $NPV \leftarrow 0$
(2) for each level l_i in Input
(3) for each edge (u, u') at level l_i
(4) $NPV[(l_i, label(u), label(u'))] \leftarrow NPV[(l_i, label(u), label(u'))] + 1$
(5) return NPV
}

Fig. 6. Tree Projection



(a) A query and a stream graph (b) NNTs and their projections

Fig. 7. Illustration of NNT Projection

After converting each NNT to its NPV, a graph G is now represented as a set of vectors $G = \{NPV(u_1), NPV(u_2), \ldots, NPV(u_n)\}$. According to Lemma 4.1, we have the following necessary condition if a graph Q is subgraph isomorphic to graph G.

Lemma 4.2: Given two graphs Q and G, if Q is subgraph isomorphic to G, for each node $u \in Q$, there must exist node $v \in G$ such that $NPV(v)$ *dominates* $NPV(u)$ (denoted as $NPV(u) \preccurlyeq NPV(v)$. Here "dominates" means the value of each dimension in $NPV(v)$ is no less than that in $NPV(u)$.

According to Lemma 4.2, if one node u in Q cannot find a vertex v in a stream graph G satisfying $NPV(u) \preccurlyeq NPV(v)$, we can safely remove this pair (Q, G) from the result subgraph isomorphism candidate set without introducing false negatives. Thus, for joining streams with queries, our strategy is to use Lemma 4.2 to prune as many pairs as possible. Though the proposed projection offers an efficient comparison solution between two NNTs, the projected vectors encounter a high dimensionality problem. For example, the projected vectors could have h^2l dimensions where h is the number of distinct labels and l is the maximum depth of node-neighbor tree. In practice, this value could be large, i.e., hundreds to thousands. Such high dimensional vectors are of course unaffordable and intractable in stream settings. Fortunately, by taking a closer look at the projected vectors, we find that although we project NNTs into a high dimensional space, for each projected vector, most of its dimensions have zero values. To take advantage of this sparsity, we store the projected vectors in a compact form. In other words, we only store non-zero entries.

B. Search in the vector space

After projecting NNTs to their NPVs, we can check every possible joinable pair of streams and query graphs based on the dominant relationship of NPVs using a nested loop algorithm. We set this nested loop algorithm as the baseline and propose two improved search strategies in following subsections. Since dominant relationship refers to node projected vectors, and also vectors correspond to points in the space, in the following sections, "node u", "vector u", and "point u" are used interchangeably to refer to the $NPV(u)$.

1) Dominated set cover method: The first improved method untilizes the idea of checking the dominated vector set as a whole instead of checking the dominant relationship pair by pair. We give the formal justification in the following Theorem.

Theorem 4.1: Given two graphs Q and G, if Q is subgraph isomorphic to G, then the union set of query vectors dominated by G covers all node projected vectors in Q.

Proof Use lemma 4.2. If a query graph Q is subgraph isomorphic to G, for each node u in Q, there must exist a node v in G, where $NPV(u) \leqslant NPV(v)$. Enumerate all the dominated vectors of query Q and union them will get the dominated cover set of Q. Since each vector in Q is dominated by a vector in G, the union of all the dominated vector, covers all node projected vectors of Q. □

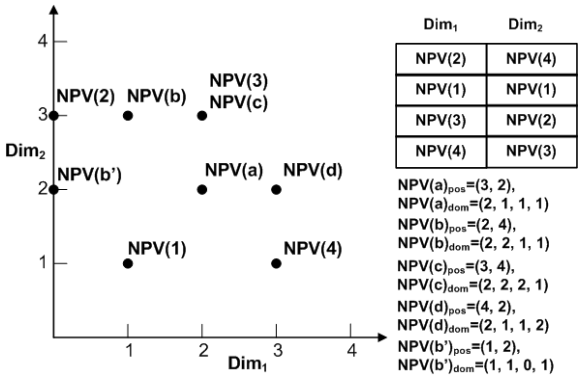
Given a query graph Q and a stream graph G, to get the dominated vector set for G, we can first compute the dominated vectors for each node of G, then union the dominated vectors of all the nodes of G to get the dominated vector set. This simple method works, yet not efficient, especially in the graph stream scenario. Thus, we propose an efficient and incremental updatable method for joining streams with queries. The basic idea is that we project all the vectors of query graph Q to their corresponding distinct single dimensional spaces (i.e. the values of vectors are not 0 in those dimensions) and sort them in each dimension. When a stream graph G comes, we project each vector of G to its single distinct dimensional space and count the number of vectors of Q that G dominates in that space. Then, we merge the results from each single dimension to generate the dominated vector set of Q. Only when all the vectors of Q are contained in the dominated set, we consider Q and stream graph G as a result candidate pair. Most importantly, in the proposed method, when the stream graph of the next timestamp comes, for each dimension, we only need to update the number of dominated vectors of Q when the position of a projected node vector in G changes. The detailed steps are listed in Procedure Dominated_Set_Cover_Join in Figure 8. k_1 and k_2 are the numbers of stream graphs and query graphs respectively.

Procedure Dominated_Set_Cover_Join {
Input: stream graphs $\{G_1, \ldots G_{k_1}\}$, query graphs $\{Q_1, \ldots Q_{k_2}\}$
Output: Reported positive pairs
(1) for $i \leftarrow 1$ to k_1
(2) for $u \in G_i$
(3) for each non-zero dimension of $NPV(u)$ {
(4) update u's position counter $NPV(u)_{pos}$
(5) update u's dominant counter $NPV(u)_{dom}$
(6) mark query vectors dominated by G_i based on $NPV(u)_{dom}$ }
(7) for $j \leftarrow 1$ to k_2
(8) if G_i dominates all vectors in Q_j
(9) $answer \leftarrow answer \cup (i, j)$
(10) return $answer$

Fig. 8. Dominated Set Cover Method

In Procedure Dominated_Set_Cover_Join (Figure 8), a few additional data structures are used to record the dominant information. First we need to store sorted vectors of a query graph Q in each distinct dimension. For example, we project the query vectors $\{NPV(1), NPV(2), NPV(3), NPV(4)\}$ of Q in Figure 9 (a) into 2 single dimensions, i.e. Dim_1 and Dim_2 as shown shown in Figure 9 (b), After sorting vectors in their increasing order in each dimension, we get $\{NPV(2), NPV(1), NPV(3), NPV(4)\}$ and $\{NPV(4), NPV(1), NPV(2), NPV(3)\}$ for Dim_1 and Dim_2, respectively. Second, for each vector of a stream graph, we maintain two counter vectors for it, namely the position counter vector and the dominant counter vector. Position counter records the stream vector's position at each single dimension. For example, for node $NPV(b)$ of G in Figure 9 (b), it is no less than the first two query vectors $\{NPV(2), NPV(1)\}$ at Dim_1 and $\{NPV(4), NPV(1), NPV(2), NPV(3)\}$ at Dim_2. Thus in $NPV(b)$'s position counter $NPV(b)_{pos}$ for Dim_1 and Dim_2

are 2 and 4, respectively, which indicates that $NPV(b)$ dominates 2 vectors at Dim_1 and 4 vectors at Dim_2 . The dominant counter vector has N entries, each for a query vector. The value in an entry indicates how many dimensions that a vector of stream graph has dominated the corresponding query vector so far. In Figure 9 (b), the dominant counter vector of $NPV(b)$, $NPV(b)_{dom}$, has 4 entries, each for a query vector. $NPV(b)_{dom}$ shows that $NPV(b)$ dominates node $NPV(1)$ twice, $NPV(2)$ twice, $NPV(3)$ once, and $NPV(4)$ once. Since the full space in the example has only two dimensions and query vectors $NPV(1)$ and $NPV(2)$ are dominated by $NPV(b)$ in the two dimensions, we can infer that query vectors $NPV(1)$ and $NPV(2)$ are dominated by $NPV(b)$ at the full space. To illustrate incremental update of Procedure Dominated_Set_Cover_Join, we assume that node b changes to b', thus b's $NPV(b)$ moves to $NPV(b')$, we can simply update its position counter vector by decreasing/increasing its position for Dim_1. Then, the dominant counter vector of b becomes $\{1, 1, 0, 1\}$. Since no entries have value 2, we infer the updated b' will not dominate any query vectors.



(a) NPVs of Q and G (b) Position and dominant counters for G

Fig. 9. Illustration of dominated set cover method

We analyze the space cost of Procedure Dominated_Set_Cover_Join (Figure 8). For maintaining sorted order in each dimension, we do not need to consider vectors having zero entries on that dimension. We define function $non - zero(i)$ to denote the number of vectors having non-zero entries on dimension i and L as the total number of dimensions in the vector space, then the space cost is:

$$\sum_{i=1}^{L} non - zero(i) = \sum_{j=1}^{k_2} \sum_{u=1}^{|Q_j|} j.L_u \approx \bar{L} \sum_{j=1}^{k_2} |Q_j| = \bar{L}N$$

Here $|Q_j|$ is the size of a query graph j and $j.L_u$ is the number of non-zero dimensions of vertex u in graph j, and $\bar{L}$ is the average number of non-zero entries for a vector, and N is the total number of vertices in query graphs.

Instead of maintaining all $L + N$ counters for each stream graph node, we only maintain counters of query nodes it encounters in its non-zero entries so far. In summary, by using compact form, the total space cost is improved to $O(\bar{L}M + \bar{N}M + \bar{L}N)$, where $\bar{N}$ is average number of encountered query nodes and $\bar{L}$ is average number of non-zero entries for a vector and M is the total number of vertices in stream graphs. Both $\bar{L}$ and $\bar{N}$ will be much smaller than L and N respectively. In terms of the time complexity, it is easy to see that the complexity of a nested loop algorithm is the lower bound of this Domincate_Set_Cover_Join procedure.

2) Skyline with early stop join method: In previous subsection, we proposed an improved method by checking whether the dominant vector set for a stream graph covers all the query vectors. Now we consider the complement problem of dominated set cover problem, search query vectors that can not be dominated by any vector of the stream graph. In other words, we want to find the *skylines* for stream graph vector set from the query vectors. In order to well illustrate the skyline-based method for joining graph streams and queries, we first briefly review the definitions of skylines as follows:

A *Monochromatic Skyline*, or Skyline, is defined as follows:

Definition 4.3: Given a set V of vectors in a $|D|$-dimensional space D, the skyline of D returns the vectors that are not dominated by the others. Here, vector v dominates tuple v', if it holds that: 1) $v[i] \leq v'[i]$ (no worse than) for each dimension $i \in D$, and 2) $v[j] < v'[j]$ (better than) for at least one dimension $j \in D$. [14]

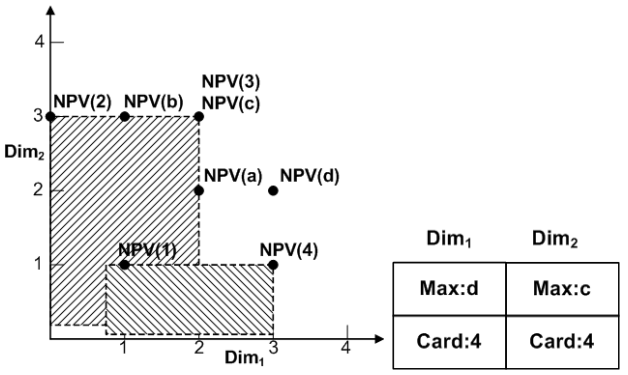
The traditional skyline is proposed for single type data set. Recently, *Bichromatic Skyline* for two types of data sets is proposed, which is defined as:

Definition 4.4: Given two $|D|$-dimensional space D and D', the bichromatic skyline of D with regard to D' returns vectors in D that cannot be dominated by any vectors in D'.

In this paper, to adapt to our join problem between stream vector sets and query vector sets, we use the "dominant" definition stated in Lemma 4.2, that is, given two node projection vectors, $NPV(u)$ and $NPV(v)$, if for each dimension $NPV[v] \geq NPV[u]$, $NPV(v)$ dominates $NPV(u)$, denoted as $NPV(u) \preccurlyeq NPV(v)$.

Based on the above skyline definitions, we propose a method to find bichromatic skyline points from query vector set with regard to stream vectors. Since once we find one skyline point in the query vector set with respect to stream vector set, we can conclude that the query graph Q is not a subgraph of the current stream graph G and remove the investigating pair from the result candidate set. Therefore, we called the proposed method skyline with earlystop join. The naive way to find a skyline point with regard to the stream vector set is to compare all query vectors with all stream vectors, as long as we find one skyline point, we stop the enumeration. This naive solution is of course inefficient. Thus, we exploit three optimization techniques to speed up searching for a skyline point. The three techniques can be categorized as query side optimization and stream side optimization.

First of all, we observe that we do not need to check all query vectors for skylines. For example, using the same query and stream graph in Figure 9, Figure 10 (a) shows that query vector NPV(3) dominates query vectors NPV(1) and NPV(2). If query vectors NPV(1) or NPV(2) is a skyline with respect to the stream vectors, due to the transitive property of



(a) skylines of NPVs from Q (b) Counters maintained for NPVs in G

Fig. 10. Illustration of skyline with early stop method

dominant relationship, query vector NPV(3) will be a skyline point as well. Furthermore, if there are some stream vectors dominate query vector NPV(3), they will dominate query vectors NPV(1) and NPV(2) as well. The above observation are formalized into the following theorem:

Theorem 4.2: Given a query graph Q and a stream graph G, if Q is subgraph isomorphic to G, all monochromatic skyline vectors of query vector set of Q will be dominated by some vectors of stream graph G.

Proof. According to Lemma 4.2, for any node u in Q, its vector $NPV(u)$ will be dominated by a node vector from G, $NPV(v)$, if Q is subgraph isomorphic to G. Thus, for any monochromatic skylines of Q, they will be dominated by some vectors of stream graph G. □

According to Theorem 4.2, we propose the first optimization: Off-line compute the monochromatic skyline points of query graph vector set, and only check whether these points are only bichromatic skylines in the stream vector set. Since the query set is fixed, we can offline compute the skyline points for this set and ignore the cost overhead of this preprocessing. Take points in figure 10 (a) for example, after preprocessing, we obtain the monochromatic skyline point set of query vectors as $\{NPV(3), NPV(4)\}$. Thus we only need to check whether these 2 points are bichromatic skylines of the stream vector set instead of checking all the 4 points.

The second optimization is to adjust the order in which the query points are processed. For example, if we always start from checking query point that is least possible to be a bichromatic skyline point with regard to the stream points, no early pruning would be possible. Our idea here is that in order to maximize the possibility of reaching an early stop, we first sort the query points based on some simple criteria such that we often can check points with high possibility of being skyline points first. Here we simply sort the query points by the maximal value of their non-zero dimensions. The rational of this heuristic is that a point with a larger value on one dimension will have less chance to be dominated by other points on the same dimension.

The optimizations above are based on the query side. After reordering the monochromatic skyline points of query vectors, we need to enumerate these points in the pre-sorted order and check whether at least one of them is a bichromatic skyline point with regard to the stream vectors. We optimize at stream side to further reduce the search cost.

The third optimization is that we will not enumerate all vectors in the stream, but only conduct subspace search within the non-zero dimensions of the query vector. An important observation is that a stream vector u will potentially dominate a query vector v if and only if u dominates v in at least one of v's non-zero entries. Take figure 10 (a) for example. When we are checking whether $NPV(3)$ is a bichromatic skyline of stream vector set, the vectors that can potentially dominate $NPV(3)$ is $NPV(d)$ and $NPV(c)$, since $NPV(d)(NPV(c))$ has larger (equal) value in Dim_1 (Dim_2) than $NPV(3)$. Finally, vectors in stream vector set that can dominate $NPV(3)$ is $NPV(c)$. Since we only conduct subspace search within non-zero dimensions of query vectors, the searching cost can be reduced. To facilitate this optimization, we project stream vectors into their corresponding single dimensions. After projection, we check whether the skyline point u of the query vector set is the bichromatic skyline for the stream vector set by choosing one of u's non-zero dimension and comparing u to all stream vectors appear in the selected dimension. Furthermore, when we choose one dimension of stream vectors to access, we heuristically pick up a dimension potentially has as few vectors to compare as possible. As shown in figure 10 (b), for each dimension, we maintain maximal value and cardinality of stream vectors in this dimension. In a certain dimension, if a query skyline node has value larger than the max value, this query vector is a skyline point and we filter out the current checking pair. For example, in figure 10, if we want to verify query point $NPV(4)$ is a bichromaitc skyline of the stream vector set or not, we search in the two dimensions and find $NPV(4)$ has value no larger than max value of each dimension. Thus, we have to use the cardinality of this dimension to estimate the number of vectors potentially need to be compared with query vector. Among all non-zero dimensions of a query vector, we take the dimension with minimal cardinality and compare query vector to vectors only appear in this dimension. When there is a tie as shown figure 10 (b) that both Dim_1 and Dim_2 have cardinality of 4, we arbitrarily choose one dimension and check all stream vectors in that dimension. The detailed steps of skyline with early stop join method are given in Procedure Skyline_with_Earlystop_Join (Figure 11).

To analyze the complexity of procedure in figure 11, we observe in the worst case, we have to enumerate all query vectors and compare with all stream vectors. Thus the lower bound of this procedure is $O(\bar{L}MN)$. In average case, with early stop, the procedure will skip many comparisons if most of the queries do not appear in streams. In such case , the complexity is estimated as $O(\bar{L}\bar{M}N)$, where $\bar{M}$ represents the average of minimal candidate sizes.

Procedure Skyline_with_Earlystop_Join {
Input: stream graphs $\{G_1, ...G_{k_1}\}$, query graphs $\{Q_1, ...Q_{k_2}\}$
Output: Reported positive pairs
(1) for each query graph Q_j
(2) for each stream graph G_i
(3) for each monotonic skyline node $u \in Q_j$ {
(4) check counters of non_zero Dim of $NPV(u)$
(5) if exist Dim_k, $NPV(u) > Max$
(6) $NPV(u)$ is a skyline to stream vectors, break
(7) else { select Dim_k with minimum $Card$ to evaluate
(8) if $NPV(u)$ is skyline to stream, break}
(9) if Q_j's skyline nodes are not skylines to G_i
(10) $answer \leftarrow answer \cup (i, j)$
(11) return $answer$

Fig. 11. Skyline with Early Stop Join Method

V. EXPERIMENTS

In this section we report the experimental results of effectiveness and efficiency tests on our method. We compare with GraphGrep[17] and gIndex[24]. We have the following results:

1) Our method has comparable high effectiveness as gIndex on static real and synthetic datasets, and is better than that of GraphGrep.
2) Our method has comparable high effectiveness as gIndex on stream real and synthetic datasets, and is more efficient than gIndex under such settings.
3) Our method scales well when the number of streams and graphs increases.

We have done extensive experiments on both static and stream datasets as well as real and synthetic datasets. For static datasets, we use a real dataset and a synthetic dataset:

1) The real static dataset we use is an AIDS Antiviral Screen Dataset containing chemical compounds. This dataset is available to the public on website of the Developmental Therapeutics Program[1].
2) The synthetic static dataset we use is generated by a generator provided by [12]. The tool can generate graphs with a specified expected average size and a specified number of distinct labels.

For stream datasets, we use real and synthetic datasets:

1) The real stream dataset is from a project in MIT Media Lab, namely the Reality Mining Dataset[2]. It contains the cell phone communication information of a certain group of MIT people during Jan. 2004 and May 2005. We use a subset called Device Span Dataset.
2) The synthetic stream dataset we use is again based on graphs generated by the tool provided by [12]. Based on the generated graphs, we will randomly generate a group of graph streams.

All experiments are conducted on a machine with Intel core 2 quad 2.4GHz * 4 CPU and 2GB memory, running on Fedora Core 7 Linux operating system. We use a few short terms to refer to several methods. They are: NPV for Node Projected Vector method of our Node-Neighbor Tree feature, gIndex for [24]'s method, GGrep for [17]'s method; NL for nested loops algorithm, DSC for dominated set cover method, Skyline for skyline with early stop method.

A. Experiments on Static datasets

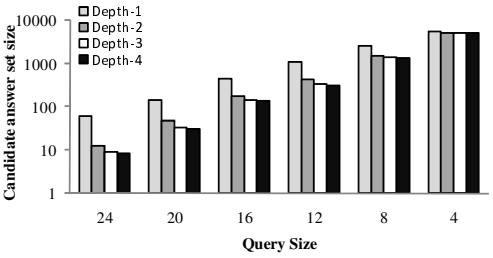
We use static datasets to test our feature's general effectiveness on graph search. For the AIDS dataset, we randomly select 10,000 graphs from the whole dataset. The selected sample has average number of nodes 24.8 and average number of edges 26.8. We use six set of queries, each of which has 1,000 queries. We denote a query set as Q_m where m means that queries in this set are connected size-m graphs extracted randomly from the dataset. We test effectiveness by using $Q_4, Q_8, Q_{12}, Q_{16}, Q_{20}, Q_{24}$, the same setting used in [24]. The synthetic dataset is generated as follows: first, a set of S seed fragments are generated randomly, the size of which is determined by a Poisson distribution with mean value I. The size of each graph is a Poisson random variable with mean T. Seed fragments are then randomly selected and inserted into a graph one by one until the graph reaches its size. Detailed description about the generator is in [12]. We generate a dataset by following parameters as suggested in [24]: D=10,000, L=200, I=10, T=50, V=4, E=1. Here D means the number of graphs to generate, L means the number of frequent patterns as possible frequent graphs, I means the average size of frequent patterns, T means the average graph size, and V means the number of distinct node labels, and E means the number of distinct edge labels. We also generate the query sets with the same parameter except that we change T to a smaller value and change D to 1,000.

1) Maximum depth of node-neighbor tree: Before comparing to other methods, we first conduct self test on the selection of maximum depth of the node-neighbor tree. In figure 12(a) and 12(b), the y-axis represents the size of candidates after filtering. We can see that using depth more than 3 will not help much in reducing the size of candidates. Thus we conclude that it suffices to use depth at most 3 for effective filtering.

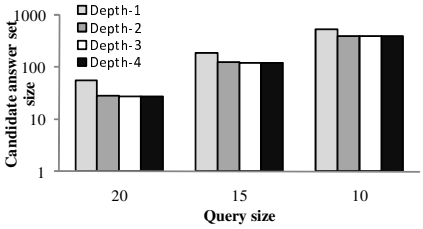
2) Effectiveness test on static data: Based on the previous results, we fix the depth to 3 and continue to compare the effectiveness with GraphGrep and gIndex.

We set following parameters for GraphGrep and gIndex: For GraphGrep we use default setting for index construction. Thus, GraphGrep indexes all paths up to length 4. The reason is that it takes too long for GraphGrep to finish enumerating paths at a longer length, i.e., 10. For gIndex, we have two settings, "gIndex1" is to set the maximum fragment size $maxL$ to be 10; and the maximum support Θ to be $0.1N$. These are the default parameters in [24]. Another setting of gIndex, namely the "gIndex2" setting is to set maximum fragment size to be 3, thus the support becomes 1 according to [24], i.e., indexing all structures up to size 3. This setting is for better running time and is used in stream dataset only.

Figure 13(a) demonstrates the comparison of performance in the AIDS dataset. We can see our method has comparable pruning power as gIndex, and is much effective than GraphGrep, which will report many false positives. Then we compare the effectiveness on the synthetic dataset. We can see from figure 13(b), again our feature has comparable pruning power as gIndex and is much more effective than GraphGrep.

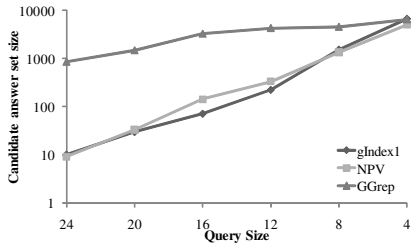


(a) Varying depth on AIDS data

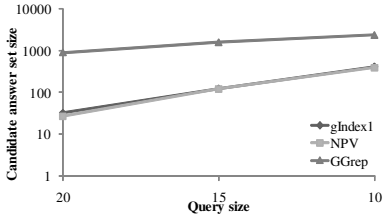


(b) Varying depth on syn. data

Fig. 12. Selection of depth

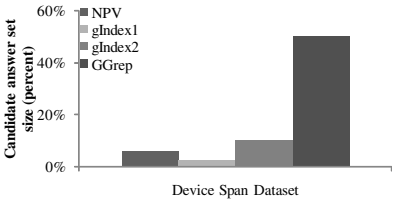


(a) Comparison on AIDS data

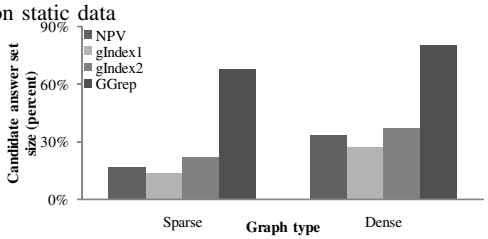


(b) Comparison on syn. data

Fig. 13. Comparison of effectiveness on static data

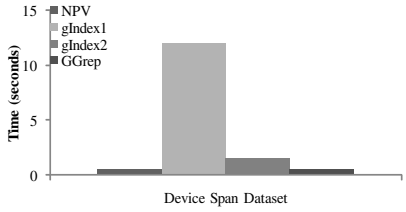


(a) Effectiveness on real data

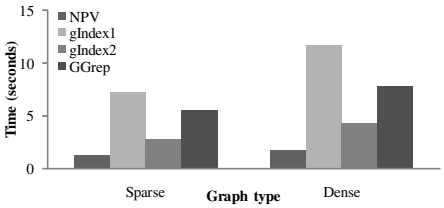


(b) Effectiveness on syn. data

Fig. 14. Comparison of effectiveness on stream data

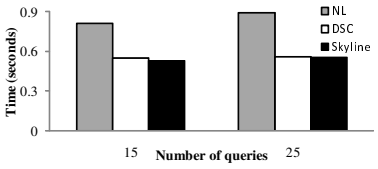


(a) Efficiency on real data

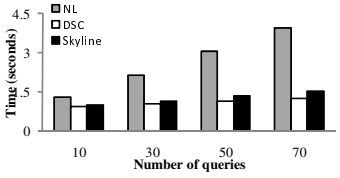


(b) Efficiency on syn. data

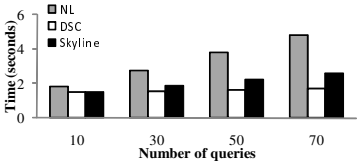
Fig. 15. Comparison of efficiency on stream data



(a) Varying queries on real data



(b) Varying queries on syn. sparse data



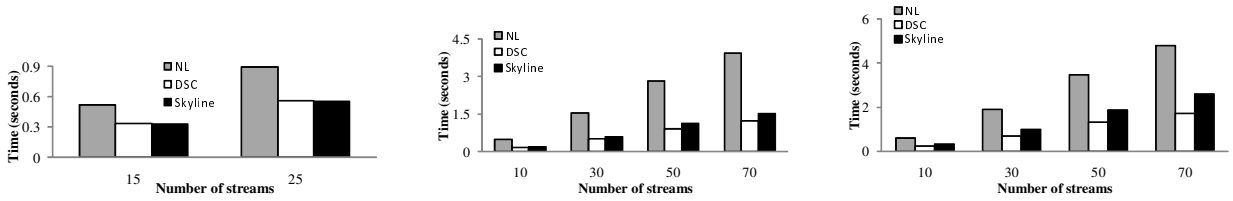
(c) Varying queries on syn. dense data

Fig. 16. Varying queries on stream data

B. Experiments on Stream datasets

We test on effectiveness and efficiency of our methods on both real and synthetic stream datasets.

For the real dataset, we use a subset of the Reality Mining Dataset from MIT Media Lab[2]. The dataset records the cellphone activity for m=n=97 users from two different labs in MIT. The records are ranging from Jan 2004 to May 2005. For our experiment, we extract the Device Span subset of the data. Device Span dataset is collected on the same 97 users whose cellphones periodically scan for nearby phones and computers over Bluetooth. We convert the data into graphs by the following way: We group all proximity information



(a) Varying number of streams on real data (b) Varying number of streams on syn. sparse data (c) Varying number of streams on syn. dense data

Fig. 17. Varying number of streams on stream data

by days. In total we have 300 distinct groups thus we have a stream graph of 300 timestamps. In a certain timestamp, a person is represented by a vertex in the graph and there is an edge between two vertices if in that day there exists proximity information record for that two people. We generate queries as follows: we randomly pick up graphs among the series having size of edges between 20 to 40. Any two people whose id values are congruent modulo 10 to each other will be labeled into the same type. By such way, we derive a graph stream with 10 kinds of distinct labels. To simulate multiple streams, we randomly reorder the original graph series to derive new graph streams. We generate 25 query graphs and 25 streams.

For synthetic dataset we first generate graphs using the tool provided by [12]. We set the parameters to D=70, L=20, I=10, T=40, V=4, E=1 to generate 70 basic query graphs. Then for each query graph we increase its number of vertices to 1.5 times of its original size by adding randomly labeled vertices into it. For each derived graph, we assign probability p_1 as edge appearing probability and p_2 as edge disappearing probability for each vertex-vertex pair. After that, we randomly flip a biased coin base on the appearing/disappearing probability for each vertex-vertex pair for 1000 timestamps, resulting a 1000 timestamp graph stream. Thus in total we have 70 streams each of which has 1000 timestamps. Setting different values of p_1 and p_2 will affect the property of stream graph significantly. We set $p_1 = 20\%$ and $p_2 = 15\%$ for generating dense graph and $p_1 = 10\%$ and $p_2 = 30\%$ for generating sparse graph. We will conduct experiments on both sparse and dense situations.

1) Effectiveness test of stream data: Figure 14 illustrates the average percent of candidate sizes reported by the three methods. Here we use our dominated set cover method to compare with the other two. In this experiment we set the number of queries and number of streams to the maximum values in their dataset, i.e., 25 queries and 25 streams for the real dataset and 70 queries and 70 streams for the synthetic dataset. According to the figure, we can see that GraphGrep has a very large candidate size. GraphGrep will report half of all stream-query pairs as candidates. On the other hand, our method and gIndex perform quite well. Here gIndex2 means the setting for better running time and gIndex1 represents the setting for better effectiveness. For the test on the real dataset, our method reports less than 6% data as candidates, while gIndex1 and gIndex2 report around 3% and 10% data as candidates, respectively. For sparse and dense synthetic datasets, our method reports around 17% and 30% respectively, while gIndex1 reports 14% and 26% and gIndex2 reports 22% and 36% respectively. The reason gIndex1 has such good performance is that it conducts frequent graph mining to extract powerful filtering features up to size 10. Our method is fast and has comparable good effectiveness.

2) Efficiency test of stream data: Figure 15 illustrates the average cost per timestamp for GraphGrep, gIndex and our method to process the stream graph search. Here we use our dominated set cover method to compare with the other two. And in this experiment we also set the numbers of queries and streams to maximum in their dataset. We can see that gIndex1 is much more costly than the rest of the methods. The reason is that gIndex has to perform mining the stream graphs at each timestamp. The mining cost is so high that gIndex becomes very inefficient in such settings. gIndex2 runs faster. However, in such settings the effectiveness goes down a lot. Both our method and GraphGrep have low cost as well due to not performing graph mining to generate feature structures.

We test the scalability of our two stream-query joining methods by varying the number of queries and the number streams. Figure 16 demonstrates average processing cost per timestamp for different number of queries when the number of streams is fixed to maximum. We can see the processing cost increases as we increase the size of queries. Also, we can see that increasing the number of queries will not affect the overall cost much for our proposed two joining methods. Figure 17 illustrates average processing cost per timestamp for different number of real and synthetic (sparse and dense) graph streams when the number of queries is fixed to the maximum. We can find that with the increase of the number of streams, the processing costs of our two proposed methods increase linearly on all the data sets. We can also observe that dominated set cover method achieves the best performance among the three compared methods in synthetic dataset and skyline with early stop method is better in the real dataset. The reason is that in the synthetic dataset, the stream graphs are relatively denser than stream graphs in the real dataset. This indicates that each vertex's NPV of the query graphs in the synthetic dataset is always dominated by some vertex's NPV in the synthetic stream graphs. In other words, the chance of earlier stop for the skyline method is small.

VI. Related Work

As a powerful modeling tool, graph has been applied in various fields for many applications. A lot of interesting works have been done in general subgraph search problem. [17] proposes GraphGrep to enumerate paths as indexing features

for filtering. In the paper a maximum path length controls the size of all possible path features. Because of the limitation of path features, [24] proposes gIndex to index frequent subgraphs as filtering features. Because of anti-monotonicity, once a subgraph pattern is not frequent, any supergraph that contains it will not be frequent as well. In query processing, the paper uses a cost model to estimate which index entries to retrieve. [8] proposes closure-tree to organize graphs into a tree based multi-dimensional index and used graph closures as bounding box. In verification of the existence of a subgraph pattern, it uses maximum matching to refine candidate vertex set. The method's pruning power is very effective, but the cost of conducting maximum matching at each iteration is relatively high. [11] uses line, circle and star as the basic features and converts the graph query task into subsequence matching problem. It is very suitable for particular domain usage, such as searching chemical compounds. [27] uses frequent subtrees as features and use central distance to further prune candidate size. In query processing, the query graph is partitioned into several frequent subtrees. By checking candidate graphs in those subtrees' index entries, the paper verifies subgraph isomorphism by reconstructing the graph using these subtree partitions. [23] decomposes graph into full set of subgraphs and index the hash value of canonical forms of the subgraphs. One drawback of this method is scalability since the size of index may be exponential. [19] index fragments of graph and join the retrieved part for query fragments. It is an approximate graph matching method. [4] uses frequent subgraphs as indexing features. Frequent graph queries are answered directly. Only infrequent queries need to be verified for subgraph isomorphism.[28] found that indexing frequent tree patterns plus a few subgraph patterns, subgraph search performance could be better than indexing all subgraph patterns. [25] proposes partition based index and search for superimposed graph search. [29] proposes GCoding for graph search. Its effectiveness is high compared to other methods. However the computation of eigenvalue features is too costly for stream setting. [26] discusses feature selection for substructure similarity search. The optimal feature selection is proved by the paper NP-complete. Thus the paper proposes approximate algorithms for finding a good selection.[20] proposes TALE for approximate large graph pattern matching. It uses hashing to encode each node's neighborhood information. By considering matching of at most two step neighbors, it prunes many unnecessary branches. [3] proposes methods for graph containment search where given a query graph the goal is to find all graphs in database that are contained by the query. [9] proposes a general graph query language for a large graph database. [16] proposes an improved subgraph isomorphism checking method using tree features. The paper also integrates indexing with subgraph searching. Thus not only subgraph isomorphism verification time is reduced, the filtering and search time is reduced as well. Compared to all the previous works on subgraph search, our work is different because our methods are proposed to conduct approximate subgraph search over graph streams, while all the previous works focus on search over static graph databases.

VII. Conclusions

In this paper, we have studied the problem of continuous subgraph search over graph streams. We have proposed an effective light-weight feature structure node-neighbor tree for effective filtering. We further proposed a novel encoding method to transform node-neighbor trees into numerical vectors and transform the approximate subgraph search problem to dominant relationship checking in the vector space. Moreover, we have proposed two different strategies to efficiently search in the space. Extensive experiments have verified the efficiency and effectiveness of our proposed methods.

Acknowledgment

Funding for this work was provided by Hong Kong RGC Grant No. 611608 and National Natural Science Foundation of China (NSFC) under Grant No. 60763001 and Grant No. 60736013.

References

[1] http://dtp.nci.nih.gov. *Developmental Therapeutics Program*.
[2] http://reality.media.mit.edu. *Reality Mining Dataset*.
[3] Chen et al. Towards graph containment search and indexing. *VLDB '07*.
[4] Cheng et al. Fg-index: towards verification-free query processing on graph databases. *SIGMOD '07*.
[5] Fortin et al. The graph isomorphism problem. *Department of Computing Science, University of Alberta*.
[6] Gao et al. Continually evaluating similarity-based pattern queries on a streaming time series. *SIGMOD '02*.
[7] He et al. Blinks: ranked keyword searches on graphs. *SIGMOD '07*.
[8] He et al. Closure-tree: An index structure for graph queries. *ICDE '06*.
[9] He et al. Graphs-at-a-time: Query language and access methods for graph databases. *SIGMOD '08*.
[10] Hulgeri et al. Keyword searching and browsing in databases using banks. *ICDE '02*.
[11] Jiang et al. Gstring: A novel approach for efficient search in graph databases. *ICDE'07*.
[12] Kuramochi M. et al. Frequent subgraph discovery. *ICDM 2001*.
[13] Lian et al. Similarity match over high speed time-series streams. *ICDE '07*.
[14] Papadias et al. An optimal and progressive algorithm for skyline queries. *SIGMOD '03*.
[15] Shamir et al. Faster subtree isomorphism. *J. Algorithms*, 33(2).
[16] Shang et al. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *VLDB '08*.
[17] Shasha et al. Algorithmics and applications of tree and graph searching. *PODS '02*.
[18] Sun et al. Graphscope: parameter-free mining of large time-evolving graphs. *KDD '07*.
[19] Tian et al. Saga: a subgraph matching tool for biological graphs. *Bioinformatics*, 23(2).
[20] Tian et al. Tale: A tool for approximate large graph matching. *ICDE'08*.
[21] Trissl et al. Fast and practical indexing and querying of very large graphs. *SIGMOD '07*.
[22] Wang et al. Dual labeling: Answering graph reachability queries in constant time. *ICDE'06*.
[23] Williams D.W. et al. Graph database indexing using structured graph decomposition. *ICDE'07*.
[24] Yan et al. Graph indexing: a frequent structure-based approach. *SIGMOD '04*.
[25] Yan et al. Searching substructures with superimposed distance. *ICDE '06*.
[26] Yan et al. Substructure similarity search in graph databases. *SIGMOD '05*.
[27] Zhang et al. Treepi: A novel graph indexing method. *ICDE'07*.
[28] Zhao et al. Graph indexing: tree + delta <= graph. *VLDB '07*.
[29] Zou et al. A novel spectral coding in a large graph database. *EDBT'08*.