Efficient search in graph databases using cross filtering
Chun-Hee Lee, Chin-Wan Chung*
Department of Computer Science, KAIST, Daejeon 305-701, Republic of Korea

Abstract
Recently, graph data has been increasingly used in many areas such as bio-informatics and social networks, and a large amount of graph data is generated in those areas. As such, we need to manage such data efficiently. A basic, common problem in graph-related applications is to find graph data that contains a query (Graph Query Problem). However, since examining graph data sequentially incurs a prohibitive cost due to subgraph isomorphism testing, a novel indexing scheme is needed.

A feature-based approach is generally used as a graph indexing scheme. A path structure, a tree structure, or a graph structure can be extracted from a graph database as a feature. The path feature and the tree feature can be easily managed, but have lower pruning power than the graph feature. Although the graph feature has the best pruning power, it takes too much time to match the graph feature with the query. In this paper, we propose a graph feature-based approach called a CF-Framework (Cross Filtering-Framework) to solve the graph query problem efficiently. To select the graph features that correspond to the query with a low cost, the CF-Framework makes two feature groups according to the query and filters out each group crossly (i.e., alternately) based on set properties. We then validate the efficiency of the CF-Framework through experimental results.

1. Introduction

Graph data has been used for a long time in computer science in order to represent various structures; many techniques related to graphs have been developed and utilized. Recently, due to technological advances, a large amount of graph data has been generated in broader areas. For example, in a social network environment, a very large social network is produced. In addition, in the bio-informatics area [12], various protein structures are modeled by a labeled graph. However, since the previous research related to graphs focuses on a small amount of data, this cannot be applied to a large amount of graph data. In the case of XML data similar to graph data, research on large-scale XML data management has been studied very actively [4,9,24,19,6,16,18]. However, XML data is primarily represented by a tree structure and XML's abundant technologies cannot be applied to graph data directly. Therefore, database communities are trying to overcome the scalability issue of graph data.

Data retrieval is a basic and common technique among the graph data management techniques. An important graph data retrieval problem is to find graph data that contains a graph query. We call this the graph query problem. The graph query problem is formally defined as follows:

* Corresponding author.
E-mail addresses: leechun@islab.kaist.ac.kr (C.-H. Lee), chungcw@kaist.edu (C.-W. Chung).

http://dx.doi.org/10.1016/j.ins.2014.06.047
0020-0255/© 2014 Elsevier Inc. All rights reserved.
• Given a graph database \( D = \{g_1, g_2, \ldots, g_n\} \) and a graph query \( q \), find all graphs \( d \in D \) such that \( q \subset d \).

**Example 1.** Suppose \( D = \{g_1, g_2, g_3, g_4\} \) and that the graph query is \( q \) as shown in Fig. 1. The answer of the query \( q \) is \( g_1 \) since \( g_1 \) is a supergraph of \( q \). In this example, we omit edge labels.

It is not possible to scan the whole data in \( D \) since subgraph isomorphism testing is an NP-complete problem [5]. To solve the graph query problem, a feature-based approach is generally used. In the feature-based approach, to find the graph data \( d \) that contains the query \( q \) (i.e., \( d \supseteq q \)), we make the feature index and compute the candidate result using the index. \((f, D_f)\) is used as an index structure, where \( f \) is some feature extracted from \( D \) and \( D_f = \{d \in D \mid f \subset d\} \). Then, given a query \( q \), the candidate answer set \((C_q)\) is computed using the precomputed list of \((f, D_f)\). If \( f \subset q \) then \( D_f \supseteq D_q \). Thus, the intersection of the \( D_f \)'s is a superset of \( D_q \), where \( f \) is a feature and \( f \subset q \). Therefore, we can compute \( C_q \) by the formula \( C_q = \bigcap_{f \subset q} D_f \), where \( F \) is a set of features. Finally, we get real answers by directly checking whether \( d \in C_q \) is a real answer. In the feature-based approach, the query processing cost is mainly affected by two kinds of costs; the index probing cost and the verification cost. The index probing cost is the cost of finding the list of features that match the query and the verification cost is the cost of checking whether a candidate answer is a real answer.

In the GraphGrep [21,8], a path is used as a feature. Although we can extract paths easily and compare two path sets extracted from the query and the graph data with a low cost, the pruning power for paths is low. This means that the index probing cost for the path feature is low, but the verification cost is very high due to the big size of the candidate answer set.

To reduce the verification cost, we can use a graph feature instead of a path feature. In the gIndex [29], a frequent and discriminative subgraph is selected as a feature. However, the index probing cost is high in the gIndex since the gIndex has to find the graph features that are contained in the query and it requires subgraph isomorphism testing.

In the FG-index [2], to remove the verification cost, the exact answer set corresponding to query \( q \) is retrieved in the index probing step. If the same feature \( f \) as the query \( q \) is found, the precomputed \( D_f \) is returned as a result. Then, query processing can be performed without the verification step of checking the candidate answer since the exact answers are matched. However, in order to compute an answer of the query without verification, the query should be in the feature set. In case the query is not in the feature set, it is not efficient to process the query. To avoid such a case, the FG-index should make the feature index for a very large number of subgraphs, which will increase the index probing cost.

In summary, in the case of using a path feature, we have the problem of low pruning power. Therefore, we have a high verification cost. In case of using a graph feature, we should find the graph features that match with the graph query. Therefore, we have a high index probing cost. To solve this dilemma, we propose the CF-Framework (Cross Filtering-Framework). Since graph features are used in the CF-Framework, we can get a low verification cost. In addition, to efficiently find the graph features contained in the query, we propose a filtering method called Cross Filtering. Using Cross Filtering, we can achieve a low index probing cost. In Cross Filtering, to find the graph features contained in the query, we make two different feature groups for the query. Then, using set properties, each group is filtered out crossly (i.e., alternately). We then perform the filtering iteratively until we search all features. During this process, we can select the graph features that correspond to the query efficiently and therefore reduce the index probing cost.

In addition, to process the graph query more efficiently, the CF-Framework provides a two-step architecture to compute candidate answers. In the first candidate answer computation, we evaluate a loose candidate answer set with a small cost using features that are easy to compute; we call them simple features. In the second candidate answer computation, we compute a tight candidate answer set using graph features and the result of the first candidate answer computation. In this step, we perform Cross Filtering.

---

1 Formally, for graph data \( g_1 \) and \( g_2 \), \( g_1 \subset g_2 \) means that \( g_1 \) and \( g_2 \) have a subgraph isomorphism from \( g_1 \) to \( g_2 \). For convenience, we say that \( g_1 \) is a subgraph of \( g_2 \) if \( g_1 \subset g_2 \). We will define the subgraph isomorphism formally in Section 3. Also, note that we use the symbol \( \subset \) instead of the symbol \( \subseteq \) to indicate “subset” (i.e., \( A \subset B \)).
1.1. Contributions

Our contributions are as follows:

- **Efficient graph feature filtering method (Cross Filtering).** Although a graph feature has a high pruning power, it is hard to find the graph features contained in a query. To overcome the problem, we propose Cross Filtering. By filtering out graph features crossly and iteratively based on the derived set properties, Cross Filtering can select graph features efficiently.

- **CF-Framework to process a graph query.** We propose an effective framework to process a graph query using simple features and graph features. We first compute a loose candidate answer set very efficiently using simple features, and then we compute a tight candidate answer set using graph features and the above candidate answer set.

- **Experimentation to validate our proposed approach.** Through an experimental study using a real dataset, we show that the CF-Framework can process graph queries efficiently. For various types of queries, the CF-Framework outperforms the previous approach in most cases in terms of the query processing time.

1.2. Organization

The rest of the paper is organized as follows. We discuss related work in Section 2 and explain the preliminary concepts and the overall procedure in Sections 3 and 4, respectively. We describe the first candidate answer computation using simple features in Section 5, then we present the second candidate answer computation using graph features in Section 6. Finally, we show experimental results in Section 7 and conclude the paper in Section 8.

2. Related work

XML data is similar to graph data. We can generally express XML data as a tree, a subset of a graph, and if we use IDREF, the XML model will become a graph. To process XML data, various indexing and query processing techniques have been proposed [4,9,24,19,6,16,18,13,28]. However, they focus on XML data that has a tree structure. Thus, the techniques studied in the XML area cannot be easily adapted to graph data. Therefore, many approaches are proposed in order to deal with graph data [21,8,29,3,38,14,11,27,37,7,15].

To process graph data efficiently, Shasha et al. [21,8] proposed a path-based approach. All paths within the maximum path length are extracted from a database and indexed. Given a query, the paths corresponding to it are retrieved and a candidate answer set is computed using the paths. However, in the path-based approach, the filtering capability of the paths will be degraded compared to that of the graph since the path loses the structural information of the graph.

As an alternative to paths, frequent subgraphs can be used as a good feature set. Subgraphs can preserve the structural information of the original graph data. However, the number of subgraphs can be very large. In the glndex [29], to reduce the size of the feature set, frequent and discriminative subgraphs are selected among many subgraphs. The key idea in the glndex is that if two features \( f_1 \) and \( f_2 \) have a subgraph relationship \( (f_1 \subset f_2) \) and similar frequencies, we do not have to keep both \( f_1 \) and \( f_2 \). Using this idea, the glndex can reduce the size of the feature set. However, the glndex has difficulty finding the feature list contained in query \( q \).

We have the high verification cost to process the graph query due to the subgraph isomorphism testing. The FG-index [2] can avoid the verification cost if the index contains the graph query. For a non-FG-query that is not contained in the index, the query performance of the FG-index is not efficient. To process many types of queries without verification, the FG-index should construct the index with a very large number of subgraphs. However, since the number of possible subgraphs is incredibly large, there may still be many non-FG-queries. The FG*-index [3] proposes an FAQ-index to solve the problem. If the non-FG-query is in the FAQ-index which is dynamically built from the set of frequently asked non-FG-queries, then the FG*-index returns the result of the non-FG-query without verification. However, the essential problem of the FG-index cannot be avoided.

In GCoding [38], a novel encoding method for graph indexing is proposed. In GCoding, graph data is encoded into graph code by combining all the vertex signatures that represent the local structure around a vertex. For effective encoding of the local structure around a vertex, the interlacing theorem for eigenvalues is utilized. However, GCoding has the limitation of representing an intrinsic property of a graph, since structural information may be lost during the encoding. Therefore, its filtering capability may be degraded compared to approaches that use frequent subgraphs.

In addition, many kinds of graph search techniques that are in contexts different from ours have been studied. Similar graph search techniques are proposed in [10,26,30,25,17,23]. The containment search technique [1] is devised using a contrast subgraph. Yuan et al. dealt with the subgraph search problem and a similar subgraph search problem in an uncertain graph database [33,34]. In [36,35,22], the subgraph search in a single large graph instead of a collection of graph data is proposed. GraphREL [20] provides a framework to store and query graph data in an RDBMS. In addition, a method to update a graph index incrementally is proposed in [32,31] while many approaches do not consider the environment in which a graph index is updated. In particular, Yang and Jin [31] consider the graph search problem for a single large graph in a dynamic
environment where nodes or edges are inserted or deleted over time. They partition the graph into core regions and extend these core regions to index regions in order to reduce the update time of the indices.

3. Preliminary

In this section, we describe notations and the basic lemma. A graph consists of vertices and edges. We can represent the graph by \(G = (V, E, L, l)\), where \(V\) is the set of vertices, \(E\) is the set of edges, \(L\) is the set of vertex and edge labels, and \(l\) is a function from a vertex or an edge to a label in \(L\). In this paper, we consider an undirected, labeled graph. In addition, we assume that both the nodes and edges have labels.

**Definition 1.** For any two graphs \(g_1 = (V_1, E_1, L_1, l_1)\) and \(g_2 = (V_2, E_2, L_2, l_2)\), we say that \(g_1 \subset g_2\) if there exists an injective function \(f : V \rightarrow V'\) such that (1) for all \(v \in V, f(v) \in V'\) and \(l(v) = l(f(v))\), and (2) for all \((v_1, v_2) \in E, (f(v_1), f(v_2)) \in E'\) and \(l((v_1, v_2)) = l'((f(v_1), f(v_2)))\). We call \(f\) the subgraph isomorphism from \(g_1\) to \(g_2\).

As mentioned previously, since the subgraph isomorphism testing is an NP-complete problem [5], we construct the index to reduce the computation of query processing. Given query \(q\), feature \(f\), and graph database \(D\), we define \(D_f\) and \(d_f\) as follows:

- \(D_f = \{g \in D | g \supset f\}\)
- \(d_f = \{g \in D | g \subset f\}\)

We can evaluate a candidate answer set basically by using \(D_f\). In addition, to compute a tighter candidate answer set, we can use \(d_f\) as well which will be explained in Section 6.3.

The following lemma can be derived easily from the definition of \(D_f\).

**Lemma 1.** For any two graphs \(g_1\) and \(g_2\), if \(g_1 \subset g_2\), \(D_{g_1} \supset D_{g_2}\).

By Lemma 1, we can compute a candidate answer set \(C_q\) by intersecting \(D_f\)'s for \(f \subset q\), where \(q\) is a query. For example, suppose that \(f_1, f_2,\) and \(f_3\) in Fig. 2 are the graph features extracted from \(g_1, g_2, g_3,\) and \(g_4\) in Fig. 1 and the query \(q\) in Fig. 1 is given. Then, \(f_1\) and \(f_2\) are contained in \(q\). Therefore, \(C_q = D_{f_1} \cap D_{f_2} = \{g_1, g_3\}\).

4. Overall procedure

In this section, we describe the overall procedure. In a graph feature-based approach, there are two major steps in processing the graph query problem, the first of which is the index construction step. In this step, it is important to extract useful graph features from a graph database. Many papers deal with a method of extracting effective graph features [29,37]; therefore, in this paper we do not focus on extracting graph features. Instead, we assume that the graph features are given.

The second step is a query processing step. Query processing consists of the first candidate answer computation, the second candidate answer computation, and the verification. In the first candidate answer computation, we evaluate a loose candidate answer set in a very short time using simple features. In the second candidate answer computation, we evaluate a tight candidate answer set using graph features and the above result.

However, since we need much time to find the graph features contained in the query in the second candidate answer computation, we propose Cross Filtering. To find the graph features efficiently in Cross Filtering, we compute Group1 and Group2 using simple features\(^2\) and the computation cost is low. Group1 and Group2 are defined as follows:

- Group1 \(\supset F_{sub} = \{f | f \in F, f \subset q\}\), where \(F\) = graph feature set, \(q\) = query
- Group2 \(\supset F_{sup} = \{f | f \in F, f \supset q\}\), where \(F\) = graph feature set, \(q\) = query

By crossly filtering out the graph features from Group1 and Group2, we can get the graph features contained in the query efficiently. In the verification, which is the final process of query processing, we verify the candidate answers to get the real answers. The detailed algorithm is shown in Fig. 5, which will be explained in Sections 5 and 6.

---

\(^2\) Simple features are used in the second candidate answer computation as well as the first candidate answer computation.
Although we have proposed simple features in this paper, these are only some of simple feature examples. In fact, we can use various graph-theoretical properties as simple features that have been proposed in graph theory communities. However, our focus is not on finding a good simple feature. In this paper, we will concentrate on the second candidate answer computation.

5. First candidate answer computation using simple features

In this section, we will explain the first candidate answer computation (Lines 2–8 of Fig. 5).

5.1. Simple features

To evaluate a candidate answer set with a small cost, we use simple features that are easy to compute. A simple feature has the following characteristics:

- **Small extraction time.** The feature extraction time from a graph should be small. If the time is long, the index construction and query processing will be performed badly.
- **Small space.** We should store the value of the simple feature with a small amount of space. If the space size of storing the simple feature is large, the index size will be large.
- **Subgraph property.** The simple feature should satisfy the subgraph property below; if the simple feature does not have this property, we cannot compute candidates through simple features. This property is the most important property for simple features.

**Definition 2 (Subgraph Property).** Let $s_f$ be a function from the graph data to some values (i.e., real numbers). If $s_f(g_1) \leq s_f(g_2)$ for any two graph data $g_1$ and $g_2$ such that $g_1 \subseteq g_2$, we say that $s_f$ has the subgraph property. The operator $\leq$ is a user-defined operator for comparing the two values. In general, the relation $\leq$ is used.

Many simple features satisfy the above three properties. As a typical example, we use the number of vertices ($n_v$) and the number of edges ($n_E$). For any two graphs $g_1$ and $g_2$, if $g_1 \subseteq g_2$, $n_v(g_1) \leq n_v(g_2)$ and $n_E(g_1) \leq n_E(g_2)$. Furthermore, $n_v$ and $n_E$ have a small extraction time and the values of those take a small space; therefore, $n_v$ and $n_E$ are simple features. According to the properties of the graph data, various kinds of good simple features exist. A good simple feature means a feature with high discriminative power. However, it is difficult to find good simple features for all kinds of graph data sets. Therefore, we can determine the simple features with the help of a domain expert.

In this paper, we do not focus on finding good simple features. Instead, we describe simple feature examples that can be used in a general environment in Appendix A.

5.2. Candidate answer computation using simple features

As explained in the overall procedure, we first compute a loose candidate answer set using simple features. We will explain Lines 2–8 of Fig. 5 in detail.

Given a query $q$, we first extract the values of the simple features of $q$, $s_f(q)$ (Line 2 of Fig. 5). The function $s_f(x)$ is defined by $[s_f(x), s_f(x), \ldots, s_f(x)]$, where $k$ is the number of simple features. Then, we evaluate the candidate answer set using the formula $C_q = \{ g \in D | s_f(q) \leq s_f(g) \}$ (Line 3 of Fig. 5). Since $s_f(q)$ is a vector, we define $\leq$ in a vector. If $s_f(g) = (v_1, v_2, \ldots, v_k)$ and $s_f(g') = (v'_1, v'_2, \ldots, v'_k)$, then $s_f(g) \leq s_f(g')$ means $v_1 \leq v'_1$ AND $v_2 \leq v'_2$ AND $\cdots$ AND $v_k \leq v'_k$, where $\leq$ is defined in each simple feature. We are sure that $C_q$ is a superset of $D_q$ because simple features have the subgraph property that states that for any two graphs $g_1$ and $g_2$, if $g_1 \subseteq g_2$, then $s_f(g_1) \leq s_f(g_2)$. If the size of $C_q$ is small, we do not proceed with the second answer set computation (Lines 4–8 of Fig. 5). If we design simple features well, we can process graph queries easily with only simple features. However, in general cases, since the discriminative power of simple features is low, we need to use graph features.

6. Second candidate answer computation using graph features

In this section, we will explain the second candidate answer computation (Lines 10–18 of Fig. 5) including the verification (Lines 19–21 of Fig. 5).

6.1. Graph features

Although simple features are easy to manage, the discriminative power of simple features may be low. Then we cannot effectively filter out graph data. It may be impossible to find simple features with high discriminative power for all kinds of
graph data sets. Therefore, we use frequent subgraphs as a good feature additionally. However, we have difficulty managing graph data. Thus, an efficient graph feature filtering method should be devised.

6.2. Candidate answer computation using graph features

To compute a tight candidate answer set $C_q$, we use graph features that are a set of special subgraphs. However, we have difficulty dealing with graph features while simple features are easy to handle. We propose a method called Cross Filtering to find the graph features contained in the query efficiently and integrate it with query processing. In the Cross Filtering, we compute the candidate answer set as well as the partial answer set. The partial answer set $P_q$ is defined by $P_q = \cup_{f \in F} D_f$, where $F$ is the feature set. Fig. 3(a) shows the relationship among $C_q$, $P_q$, and $D_q$. We do not have to verify $P_q$ because all elements in $P_q$ are in the real answer set. Therefore, we only have to check whether the elements in $C_q - P_q$ are in the real answer set. As the final result, we return (the result of verification) $\cup P_q$. The FG*-index [3] also uses a partial answer set. If the query is not in the FG*-index, the FG*-index uses the FAQ-index which is dynamically built from the set of FAQs (Frequently Asked non-FG-Queries). If the query does not match with the FAQ-index, the FG*-index finds $q$’s subgraphs and supergraphs, then computes a candidate answer set and a partial answer set. However, it is inefficient to find graph features $f_1(< q)$ and $f_2(\geq q)$ for computing the candidate answer and the partial answer in the FG*-index. In Cross Filtering, we devised a smart method to find $f_1$ and $f_2$ simultaneously.

Consider a graph feature $f$ and $D_f = \{d \in D | f \leq d\} = \{g_{i_1}, g_{i_2}, \ldots, g_{i_n}\}$. Although $D_f$ consists of graph data, we store a set of graph identifiers instead of a set of graph data. That is, we store $D_f = \{i_1, i_2, \ldots, i_n\}$, where $i_k$ is the identifier for graph $g_{i_k}$ ($1 \leq k \leq n$). Therefore, it is easier to deal with $D_f$ than $f$ since $D_f$ is a set of numbers and $f$ is a graph. Furthermore, $D_f$ exploits the characteristics of $f$ well. Thus, to avoid the high computation for graph comparison, we use $D_f$ instead of $f$ and derive some formulas related to $D_f$.

In Cross Filtering, we first compute the initial Group1 and Group2 using simple features. Since we use simple features, the computation cost to get the initial Group1 and Group2 is ignorable. This example will be shown in detail in Step 1 of Example 2.

![Fig. 3. $C_q$, $D_q$, and $P_q$.](image)
For any two graphs \( g_1 \) and \( g_2 \), if \( D_{g_1} \not\supset D_{g_2} \), then \( g_1 \not\subseteq g_2 \).

**Proof.** Lemma 2 is the contraposition of Lemma 1. □

Based on Lemma 2, we are sure that if \( D_{h_1} \not\supset D_{h_2} \), \( f_1 \) is not a subset of \( f_2 \). For example, suppose that \( D_{h_1} = \{1, 2, 3, 4, 5\} \) and \( D_{h_2} = \{1, 2, 3, 6\} \). Since \( D_{h_2} \not\supset D_{h_1} \), it is true that \( f_1 \not\subseteq f_2 \).

Consider \( f_i \) such that \( f_i \subset q \). For \( f \in F \) (\( F \): graph feature set), if \( D_{h_i} \not\supset D_f \), we can say that \( f_i \not\subset f \). Then, what relationship do \( q \) and \( f \) have?

**Lemma 3.** If \( f_i \subset q \) and \( D_{h_i} \not\supset D_f \), then \( q \not\subset f \).

**Proof.** By Lemma 2, \( f_i \not\subset f \). Since \( f_i \not\subset f \), there exists \( a \in f_i \) such that \( a \neq f \) as shown in Fig. 4(a). Since \( f_i \subset q \), \( a \in q \). By \( a \in q \) and \( a \neq f \), \( q \not\subset f \). □

In the same way, we can consider \( f_i \) such that \( f_i \supset q \). For \( f \in F \), if \( D_{h_i} \not\subset D_f \), \( f_i \not\supset f \). What relationship do \( q \) and \( f \) have in this case?

**Lemma 4.** If \( f_i \supset q \) and \( D_{h_i} \not\subset D_f \), then \( q \not\supset f \).

**Proof.** By Lemma 2, \( f_i \not\supset f \). Since \( f_i \not\supset f \), there exists \( a \in f \) such that \( a \neq f_i \) as shown in Fig. 4(b). Since \( f_i \supset q \), \( a \neq q \). By \( a \in f \) and \( a \neq q \), \( q \not\supset f \). □

By Lemma 3, if \( D_{h_i} \not\supset D_{f_i} \), where \( f_i \subset q \), then \( q \not\subset f \). Furthermore, by Lemma 4, if \( D_{h_i} \not\subset D_{f_i} \), where \( f_i \supset q \), then \( q \not\supset f \). By using the concepts, we devise an efficient graph feature filtering algorithm which can be integrated with the query processing algorithm.

In order to perform filtering for graph features, we choose one feature \( f_i(\subset q) \) from Group 1. We compute candidates using \( C_q = C_q \cap D_{h_i} \), then we reevaluate Group 1 and Group 2 as follows: Given \( f_i(\subset q) \).

- **Group 1:** \( \{ f \in \text{Group 1} | D_f \not\supset D_{h_i} \} \).
- **Group 2:** \( \{ f \in \text{Group 2} | D_f \subset D_{h_i} \} \).

We filter out a feature \( f \in \text{Group 1} \) such that \( D_f \supset D_{h_i} \), because \( D_f \) cannot reduce the size of \( C_q \) in that case (See Group 1 in Step 2 of Example 2). In addition, since if \( D_{h_i} \not\supset D_f \) then \( q \not\subset f \) (by Lemma 3), we can filter out a feature \( f \) in Group 2 such that \( D_f \not\subset D_{h_i} \) (See Group 2 in Step 2 of Example 2).

---

4 For convenience, feature \( f \) with the same simple feature values as the query is contained in Group 1.

5 Strictly speaking, we cannot use the notation \( a \in g \) for graph \( g \) since graph \( g \) is not a set. However, since a graph consists of a set of vertices and a set of edges, it can be used in our mathematical proof.
Next, we choose one feature \( f_i \subseteq q \) from Group2. \( D_f \) is a partial answer of \( D_q \) since if \( f_i \supseteq q \) then \( D_f \supseteq D_q \) (by Lemma 1). We compute the partial answer set \( P_q \) with \( f_i \) by computing \( P_q = P_q \cup D_f \). Then, using \( f_i \), we reevaluate two groups as follows:

- **Group1**: \( \{ f \in \text{Group1} \mid D_f \supseteq D_h \} \)
- **Group2**: \( \{ f \in \text{Group2} \mid D_f \subsetneq D_h \} \)

We remove \( f \in \text{Group1} \) such that \( D_f \supseteq D_h \) since if \( D_f \subsetneq D_h \) then \( q \not\supset f \) by Lemma 4 (See Group1 in Step 3 of Example 2). In a similar way to that of \( f_i \), we compute Group2. If \( D_f \subsetneq D_h \), \( D_f \) cannot increase the size of \( P_q \). Therefore, we add the condition \( D_f \subsetneq D_h \) when evaluating Group2 (See Group2 in Step 3 of Example 2).

We perform the above two procedures iteratively until we have searched all features. The subset selection (\( f_i \)) and the superset selection (\( f_s \)) are performed in turn while effectively reducing the size of each group. Through the subset selection (\( f_i \)), \( C_q \) is reduced iteratively and through the superset selection (\( f_s \)), \( P_q \) is increased iteratively as shown in Fig. 3(b).

The second candidate answer computation with Cross Filtering is summarized in Lines 10–18 of Fig. 5. In Line 10, the partial answer set is initialized. Two groups, Group1 and Group2, are made using the simple features in Line 11, we then perform the two steps iteratively until we have searched all the features in Lines 12–18. In Lines 13–15, filtering and \( C_q \) computation are performed with \( f_s \). In Lines 16–18, filtering and \( P_q \) computation are performed with \( f_i \). Finally, we do the verification on only \( C_q - P_q \) and return the result (Lines 20–21).

In the algorithm of Fig. 5, we choose a seed subgraph feature (\( f_i \)) or a seed supergraph feature (\( f_s \)) of the graph query using a sequential selection and subgraph isomorphism checking (Lines 13 and 16). We choose one element in Group1 (or Group2) and check the subgraph isomorphism between the element and the graph query. If they do not have a subgraph relationship, we choose another element. At this time, we may not abandon the information that they do not have a subgraph relationship. Using additional filtering that will be explained in Section 6.3, we can further reduce Group1 and Group2. In addition,
during the iteration of Cross Filtering, the sizes of Group1 and Group2 are continuously being reduced by filtering out unnecessary graph features.

However, we can use a heuristic selection method to reduce the time spent selecting a seed subgraph feature \( f_i \) or a seed supergraph feature \( f_i \) of the graph query. For example, we can sort graph features by the number of nodes and choose a graph feature in ascending or descending order. An elegant heuristic selection method will be our future work.

**Example 2.** Fig. 6 shows 12 graph features and their properties. In this example, we use the number of vertices and the number of edges as simple features. \( D_f \) is the precomputed index and Relationship shows the relationship between the feature and the query. \( d_f \) is used in Example 3. We assume that the initial \( C_q = \{g_1, g_2, g_3, g_4, g_5, g_6, g_7, g_8, g_9, g_{10}\} \).

**Step 1:** We make two groups, Group1 and Group2, using two simple features. Since the simple features of the query are \( (10, 12) \), Group1 = \{f_1, f_2, f_3, f_4\} and Group2 = \{f_5, f_6, f_7, f_8, f_{10}, f_{11}\}. \( f_3 \) and \( f_1 \) are filtered out.

**Step 2:** We select \( f_1(\subset q) \) from Group1 as \( f_1 \). Then, \( C_q = C_q \cap D_{f_1} = \{g_1, g_2, g_3, g_4, g_5, g_6, g_7, g_{10}\} \), then we filter out Group1 and Group2 using \( f_1 \). Since \( D_{f_1} \) in Group1 is a superset of \( D_{f_1} \) filtered out and \( f_1 \) is filtered out due to the fact that \( D_{f_1} \not\subseteq D_{f_1} \). Therefore, Group1 = \{f_2, f_3\} and Group2 = \{f_5, f_6, f_{10}, f_{11}\}.

**Step 3:** We check \( f_5 \) from Group2 in order to choose \( f_5 \). However, we remove \( f_5 \) from Group2 since \( f_5 \not\supseteq q \). Next, we select \( f_6(\supset q) \) from Group2. We compute \( P_q = P_q \cup D_{f_5} = \{g_1, g_2, g_3, g_4, g_5, g_6, g_7, g_{10}\} \). Using \( f_5 \), we reevaluate Group1 and Group2. \( D_{f_5} \) and \( D_{f_5} \) do not include \( D_{f_5} \); therefore, they are removed from Group1. In addition, \( D_{f_5} \subseteq D_{f_5} \); therefore, \( D_{f_5} \) is removed from Group2. Then, Group1 = \{\} and Group2 = \{f_5\}.

**Step 4:** We check \( f_{11} \) and remove it. Then, Group1 = \{\} and Group2 = \{\}.

**Step 5:** Since Group1 = \{\} and Group2 = \{\}, we stop filtering. \( C_q = \{g_1, g_2, g_3, g_4, g_5, g_6, g_7, g_{10}\} \) and \( P_q = \{g_1, g_3, g_4\} \). Therefore, we verify \( C_q - P_q = \{g_2, g_5, g_6, g_7, g_{10}\} \). We get \( \{g_2, g_5\} \) from the verification and return \( \{g_2, g_5\} \cup \{g_1, g_3, g_4\} \).

6.3. Additional filtering

In this subsection, we will provide tighter formulas for evaluating Group1 and Group2 than those in Line 18 of Fig. 5. Therefore, we can filter out Group1 and Group2 more effectively. In Line 16, to choose one feature \( f_i \), we should check elements in Group2 sequentially until we find \( f_i \supseteq q \). Then, we can get a list of features \( \{f_i\} \) such that \( f_i \supseteq q \) during the execution of Line 16. To provide tighter formulas, we use this information. Assume that \( NA_q \) is a non-answer set such that \( d \not\supseteq q \) for all \( d \in NA_q \). Then, we can derive the following two lemmas with respect to \( NA_q \).

![Table](image)

**Fig. 6.** Example of the second candidate answer computation.
Lemma 5. If \( D_f \cap NA_q \neq \phi, q \not\subset f \).

Proof. Consider the contraposition of this lemma. That is, if \( q \subset f \), then \( D_f \cap NA_q = \phi \). If \( q \subset f \), \( D_q \supset D_f \). Since \( D_f \) is a subset of the answer set \( D_q \), all elements in \( D_f \) are not in \( NA_q \). \( \square \)

Lemma 6. If \( D_f \subset NA_q \) and \( |D_q| \neq 0 \), then \( f \not\subset q \).

Proof. Consider the contraposition of this lemma. That is, we prove if \( f \subset q \), then \( D_f \not\subset NA_q \) or \( |D_q| = 0 \). There are two cases.

Case 1: \( |D_q| = 0 \). In this case, it is trivial.
Case 2: \( |D_q| \neq 0 \). If \( |D_q| \neq 0 \), then \( D_f \) has at least one element (i.e., one answer).

If \( f \subset q \), then \( D_f \supset D_q \). Therefore, \( D_f \) contains all answers for \( q \). Therefore, \( D_f \) has at least one answer. This means that \( D_f \not\subset NA_q \). \( \square \)

Using the above two lemmas, we can revise the computation of Group1 and Group2 in Line 18 of Fig. 5 as follows:

- Group1 := \{ \( f \in \{ \) Group1 \( \} \cup D_f \) and \( (D_f \not\subset NA_q \) or \( |D_q| = 0 ) \}\}.
- Group2 := \{ \( f \in \{ \) Group2 \( \} \cup D_f \) and \( D_f \cap NA_q = \phi \) \}.

If a partial answer set is not null, then \( |D| \) is not zero. Therefore, instead of the condition \( |D_q| = 0 \), we can use the condition \( |P_f| = 0 \) since checking the condition \( |D_q| = 0 \) is not straightforward. We can filter out graph features further. However, computing a non-answer set is not straightforward since we keep only \( D_f \). To get \( NA_q \), we compute \( d_f = \{ g \in D | g \subset f \} \) additionally.

Lemma 7. If \( q \not\subset f \), then, for all \( x \in d_f \), \( x \not\subset q \). That is, \( d_f \subset NA_q \).

Proof. When \( f \not\subset q \), there exists \( a \in q \) such that \( a \notin f \). Therefore, a subset of \( f \) does not contain element \( a \). Thus, we conclude that \( x \not\subset q \) for all \( x \in d_f \). This means that \( d_f \) is included in \( NA_q \). \( \square \)

By Lemma 7, we can calculate a non-answer set using the list of features \( \{ f_i \} \) such that \( f_i \not\subset q \). Therefore, we can replace Lines 16–18 in Fig. 5 by the algorithm in Fig. 7. While the loop from Line 16 to 18 of Fig. 7 is being performed, \( NA_q \) gradually increases.

Example 3. We show the process of the algorithm of Fig. 5 with the additional filtering. The graph features in Fig. 6 are used.

Step 1: We make two groups, Group1 and Group2. Group1 = \{ \( f_1,f_2,f_3,f_4 \) \} and Group2 = \{ \( f_5,f_6,f_7,f_8,f_9,f_{10},f_{11} \) \}.
Step 2: We select \( f_1(\not\subset q) \) from Group1 as \( f_i \). Therefore, \( C_q = \{ g_1,g_2,g_3,g_4,g_5,g_6,g_7,g_{10} \} \), and we filter out Group1 and Group2 using \( f_i \). Therefore, Group1 = \{ \( f_2,f_3 \) \} and Group2 = \{ \( f_5,f_6,f_7,f_{10},f_{11} \) \}.
Step 3: we select \( f_6 \) from Group2 as \( f_i \). However, since \( f_6 \not\subset q \), we remove \( f_6 \) and update \( NA_q \) which was originally the null set. \( NA_q = NA_q \cup D_{f_6} = \{ g_{10},g_{11},g_{12},g_{14} \} \). Then, we select \( f_5(\not\subset q) \) from Group2 as \( f_i \). We compute \( P_q = P_q \cup \{ g_1,g_3,g_5 \} \).
Then, we reevaluate Group1 and Group2 using both \( NA_q \) and \( f_5 \). \( f_1,f_2 \) and \( f_3 \) are filtered out since \( D_{f_5} \cap NA_q \neq \phi \), \( D_5 \subset NA_q \) and \( D_6 \subset NA_q \), respectively, and \( f_5 \) and \( f_6 \) are filtered out since \( D_{f_6} \not\subset D_{f_5} \) and \( D_{f_5} \not\subset D_{f_6} \). Note that \( f_2 \) and \( f_3 \) can be filtered out in the case of both \( NA_q \) and \( f_5 \) is filtered out since \( D_{f_5} \subset D_{f_6} \). Therefore, Group1 = \{ \} AND Group2 = \{ \}.

7. Experiments

In this section, we conduct experimental evaluations to show the efficiency of the CF-Framework.

7.1. Experimental environment

To evaluate the efficiency of the CF-Framework, we implement the CF-Framework and the FG-index using JAVA and conduct experiments with an Intel Core 2 Duo 2.00 GHz CPU and 3.0 GB RAM. We adapt the FG-index to our environment and since the main factor of the query performance in the graph indexing is the number of subgraph isomorphism checkings, we implement two systems on memory. In addition, to compute frequent subgraphs, we use gSpan software.\(^6\) We use the FG-index instead of the FG* index because the FG*-index is based on the FG-index and the FG*-index assumes the knowledge of query workload. The approach utilizing the query workload can be applied only in the limited environment and is not in the scope of our research. Therefore, we compare our approach with the FG-index which does not use the query workload. In

\(^6\) gSpan can be downloaded at http://www.cs.ucsb.edu/xyan/software/gSpan.htm.
addition, we do not use the best subgraph isomorphism algorithm since the subgraph isomorphism test is beyond the scope of this paper and we assume that it needs much time.

As an experimental data set, we use the data set used in gIndex [40,29]. The description of the data set is shown in Fig. 8.

The size of data is 10,000 and the number of distinct labels is 51. The average number of vertices is 25.4 and the average number of edges is 27.4. The data is extracted from AIDS Antiviral Screen Dataset [39] containing chemical compounds. Also, graph data in the data set has vertex labels as well as edge labels.

Furthermore, we use query sets Q4, Q8, Q12, Q16, Q20, and Q24 used in the gIndex. The number following Q represents the number of edges. Each query set Q-m is generated by extracting a connected m-sized subgraph from data sets. Each query set Q-m has many queries with m-edges. We use the first ten queries in each Q-m and average their results.

### 7.2. Experimental results

We measure the query response time with six types of queries (Q4, Q8, Q12, Q16, Q20 and Q24) when the size of the data is 10000. The query response time means the time to get the final answer \( D_q \) given the query \( q \). Fig. 9 shows the query response time according to the size of the query; the size of the query is the number of edges in the query (i.e., graph).

We use the minimum frequency threshold \( \sigma \) which is the parameter used for extracting frequent subgraphs. If \( |D_f| \geq \sigma |D| \), \( f \) is a frequent subgraph. Fig. 9(a) shows experiments when \( \sigma = 0.1 \) and Fig. 9(b) shows the experiments when \( \sigma = 0.05 \). When \( \sigma \) is large, the number of features becomes small. We average the execution time for ten queries in each query set (Q-m).

In Fig. 9 and Fig. 9(b), in most cases, the CF-Framework shows a better performance than the FG-index in terms of the response time. The CF-Framework filters out the unnecessary graph data through two steps. Loose candidates are computed with a small cost using simple features while compact candidates are efficiently evaluated using graph features. Therefore, the CF-Framework can process various kinds of queries efficiently and shows a good performance in most cases as shown in Fig. 9. However, in the case of Q4, the CF-Framework has a worse performance than the FG-index. We will explain the reason for this in the following experiments. The query response time does not depend on only the size of the query as shown in...
Fig. 9. Experiments according to the size of the query.

Fig. 10. Experiments according to the size of the query when the size of data is 2000 and \( \sigma \) is 0.1.
The FG-index shows a much worse performance in Q16 than that in other queries when compared to the CF-Framework. In addition, we show the query response time in Fig. 10 when the size of data is small (the size of data = 2000, \( r = 0.1 \)). The result of Fig. 10 shows a tendency similar to that of Fig. 9; therefore, we do not mention it in detail.

Figs. 12–14 show experiments according to the size of the data when \( r = 0.1 \). We conducted experiments on 2000, 4000, 6000, 8000, and 10,000 graph data. As we expected, the query response time increases for both the CF-Framework and the FG-index as the size of the data increases. Except for Q4 (Fig. 12(a)), the CF-Framework is superior to the FG-index.

In the case of Q4, the number of edges in the query is small. Therefore, queries in Query Set Q4 may be contained in the index (i.e., FG-query). In the FG-index, if a query is an FG-query, the answer is returned without the candidate verification.

<table>
<thead>
<tr>
<th>Query</th>
<th>FG-Query</th>
<th>Size of Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q4</td>
<td>4</td>
<td>1930.5</td>
</tr>
<tr>
<td>Q8</td>
<td>0</td>
<td>194.4</td>
</tr>
<tr>
<td>Q12</td>
<td>0</td>
<td>23.1</td>
</tr>
<tr>
<td>Q16</td>
<td>0</td>
<td>7.8</td>
</tr>
<tr>
<td>Q20</td>
<td>0</td>
<td>1.9</td>
</tr>
<tr>
<td>Q24</td>
<td>0</td>
<td>0.4</td>
</tr>
</tbody>
</table>

Fig. 11. Characteristics of query sets.

Fig. 12. Experiments according to the size of data (Q4, Q8).
and the FG-index shows a good performance in terms of the execution time. Fig. 11 shows the number of FG-queries and the average size of results for each query set when the size of data is 10,000 and $\sigma$ is 0.1.

Query Set Q4 contains four FG-queries (40%) and the resultant size is large compared to the other query sets. While the FG-index returns the result for the FG-queries in Query Set Q4 without verification, the CF-Framework should check whether a candidate is a real answer. Thus, for Q4, the CF-Framework has a worse performance than the FG-index. However, in most cases, queries are not FG-queries. Therefore, the CF-Framework can generally process various types of queries efficiently in comparison to the FG-index. It is not possible to index all subgraphs in order to return $D_q$ without verification in the FG-index.

Fig. 15 shows the query response time with respect to $\sigma$. As $\sigma$ increases, the number of features becomes smaller. Therefore, the feature search time will be reduced. On the other hand, the size of the candidate answer will be larger since a small number of features are included in the query. There exists a trade-off between the feature search time and the time for the candidate verification. Thus, in Fig. 15, the query processing time is not significantly affected by $\sigma$, and a particular tendency on $\sigma$ is not shown.

In addition, in order to compare the benefit for the first candidate set computation (using simple features) and the benefit for the second candidate set computation (using graph features), we conducted three experiments. One using only the first candidate set computation (“First”), one that only used the second candidate set computation (“Second”), and one that used both (“First + Second”) in Fig. 16. Fig. 16 shows the number of candidate answers for “First”, “Second” and “First + Second”. For “Second” and “First + Second”, the number of candidate answers means the size of $C_q - P_q$ ($C_q$ is the candidate answer set and $P_q$ is the partial answer set) since we do not need to check the subgraph isomorphism for the partial answer set. “Second” is much more effective than “First” as shown in Fig. 16 since the second candidate set computation uses graph features. However, the second candidate set computation can be improved if it is integrated with the first candidate set computation. Therefore, “First + Second” shows the best performance in terms of the size of the candidate answer set. As the size of the query increases, the size of the candidate answer set decreases in both “First” and “Second”. This is because the simple feature value for a large-sized query is large and a large-sized query contains many graph features.

![Fig. 13. Experiments according to the size of data (Q12, Q16).](image-url)
Fig. 14. Experiments according to the size of data (Q20, Q24).

Fig. 15. Experiments according to $\sigma$. 
8. Conclusion

Recently, the large amount of graph data used in areas such as bio-informatics and social networks has been increasing. In those areas, the graph query problem is one of the most important, but subgraph isomorphism testing is an NP-complete problem. Therefore, to process a graph query efficiently, we propose a CF-Framework to retrieve graph data that contains the query. In the CF-Framework, we use two kinds of features: Simple features and graph features. Since simple features are easy to manage, the candidate set of the query is first retrieved using simple features. Then, using the graph features, which are difficult to manage but have better pruning power than the simple features, we can effectively reduce the size of the candidate set. Since it spends much time to find the graph features corresponding to the query, we propose an efficient graph-feature filtering algorithm called Cross Filtering, based on set properties. In Cross Filtering, by making two groups and filtering each group crossly, we can efficiently choose graph features that correspond to the query. The experimental results show that the CF-Framework can process various types of graph queries efficiently.

Acknowledgement

We would like to thank the editors and anonymous reviewers for their helpful comments. This work was supported by the National Research Foundation of Korea (NRF) Grant funded by the Korean Government (MSIP) (No. NRF-2014R1A1A2002499).

Appendix A. Examples for simple features

A.1. Vertex count, edge count and max degree

We can use **Vertex count**, **Edge Count**, **Max Degree** as basics. The **Vertex Count** is the number of vertices, **Edge Count** is the number of edges and **Max Degree** is the maximum degree among all vertices. For example, the Max Degrees for $g_1, g_2, g_3$ and $g_4$ in Fig. 1 are 3, 2, 3, and 2, respectively. The values of the basic simple features are highly skewed. Therefore, the discriminative power is low.

A.2. Vertex encoding and edge encoding

To exploit the property of the vertex and edge labels well, we use **Vertex Encoding** $f_v$ and **Edge Encoding** $f_e$ as simple features. Given graph data $g = (V, E, L, k)$, we encode the set of vertices $(V)$ or edges $(E)$ into the $k$-bit array. Assume two functions $f_v : L \rightarrow \{0, 1, \ldots, k-1\}$ and $f_e : (L, L) \rightarrow \{0, 1, \ldots, k-1\}$, where $L = \cup L_i$. Since the label of edge $e = (v_1, v_2)$ can be represented by $(l(v_1), l(e), l(v_2))$, in this section, we use $l(e)$ as the meaning of $(l(v_1), l(e), l(v_2))$. Using $f_v$ and $f_e$, we compute the vertex encoding value and the edge encoding value for $V_i$ and $E_i$, respectively. We assume that $M_v^g$ and $M_e^g$ are $k$-bit arrays to store vertex encoding value and edge encoding value for graph $g$, respectively, and are initialized as zero. For all $a \in V$, we set $M_v^g[f_v(l(a))]$ to 1. Then, $f_v$ returns $M_v^g$, which consists of the $k$ bit array. In the same way, for all $e \in E$, we set $M_e^g[f_e(l(a))]$ to 1. The time complexity of computing the vertex encoding value is $O(|V|)$ and that of computing the edge encoding value is $O(|E|)$. Therefore, **Vertex Encoding** and **Edge Encoding** both have a small extraction time. In addition, to save the vertex encoding value and the edge encoding value, only $k$ bits are required for each encoding value. Finally, we should check the subgraph property of $f_v$ and $f_e$. We can define $\preceq$ as follows:

![Fig. 16. The number of candidate answers.](image-url)
For the two-bit arrays $M_a$ and $M_b$, $M_a \preceq M_b$ iff $M_a \& M_b = M_a$, where $\&$ is a bitwise AND operator.

Then, $f_v$ and $f_e$ satisfy the subgraph property.

**Proof.** Prove that for any two graph data $g_1 = (V_1, E_1, L_1, l_1)$ and $g_2 = (V_2, E_2, L_2, l_2)$, if $g_1 \subseteq g_2$, then $f_v(g_1) \preceq f_v(g_2)$ and $f_e(g_1) \preceq f_e(g_2)$.

First, consider $f_v(g_1) \preceq f_v(g_2)$. We assume that $M_{g_1}[i]$ and $M_{g_2}[i]$ are $k$-bit arrays returned by the function $f_v$ for $g_1$ and $g_2$, respectively.

There are only two values in $M_{g_1}[i]$.

- **Case 1:** $M_{g_1}[i] = 1$.
  - If $M_{g_1}[i] = 1$, then $M_{g_2}[i] = 1$ (Since $V_1 \subseteq V_2$).
  - Therefore, $M_{g_1}[i] \& M_{g_2}[i] = 1$.
- **Case 2:** $M_{g_1}[i] = 0$.
  - If $M_{g_1}[i] = 0$, then $M_{g_1}[i] \& M_{g_2}[i] = 0$ regardless of $M_{g_2}[i]$ value.

Therefore, by Case 1 and Case 2, $M_{g_1}[i] \& M_{g_2}[i] = M_{g_2}[i]$ for any $i$. That is, $f_v(g_1) \preceq f_v(g_2)$.

In the same way, we can prove that $f_e(g_1) \preceq f_e(g_2)$. $\square$

**Example 4.** We can extract the vertex encoding value for $g_1, g_2, g_3, g_4$ and $q$ using $f_v$ as shown in Fig. A.17. We extract the vertex set from graph data and encode it by the function. We assume that $k$ is 4 and the function is $A \rightarrow 0, B \rightarrow 1, C \rightarrow 2, D \rightarrow 3, E \rightarrow 0$ and $F \rightarrow 1$. Since $f_v(q) \preceq f_v(g_1) = f_v(q) \preceq f_v(g_3)$. In the same way, $f_v(q) \preceq f_v(g_3)$. Therefore, $g_1$ and $g_3$ are candidates. Edge Encoding is processed like Vertex Encoding.
References


