Compressed Feature-based Filtering and Verification Approach for Subgraph Search

Karam Gouda Mosab Hassaan
Faculty of Computers and Informatics, Benha University, Egypt
{karam.gouda, mosab.hassaan}@fci.bu.edu.eg

ABSTRACT

Subgraph search in graph datasets is an important problem with numerous applications. Many feature-based indexing methods have been proposed for solving this problem. These methods have to index too many features or select some of them in order to get an index with good pruning capabilities. None of these directions can give an effective solution to all graph indexing issues. In this paper, we propose an efficient indexing approach which improves over current feature-based methods, neither by the costly feature selection nor by explicitly indexing a multitude of features. We achieve this by compressing multiple features into one feature with some neighborhood information encoded. Neighborhood is further used to prune unmatched feature occurrences between the query and data graphs, thus cutting down the search space of subgraph matching, which significantly reduce the verification cost. We implement the approach by exhaustively enumerating small paths as features. A novel path-at-a-time verification method that benefits from the occurrences pruning method is introduced. Via an extensive evaluation on both real and synthetic datasets, we show that our approach is effective and scalable, and outperforms state-of-the-art indexing methods.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: System–Query processing

General Terms

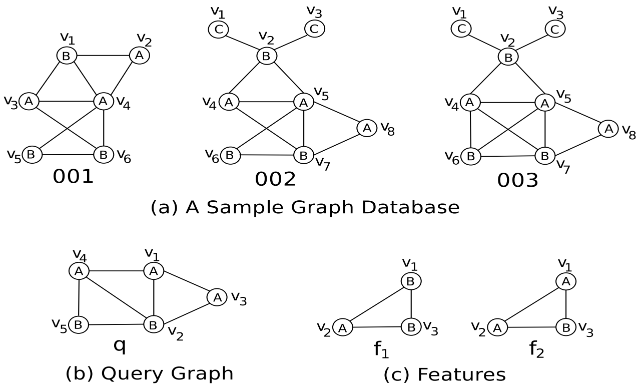
Algorithms, Experimentation, Performance

Keywords

Subgraph query processing, Feature-based indexing, Verification methods

1. INTRODUCTION

Large scale graph datasets are prevalent in many application areas such as in Bio-informatics, Chem-informatics, etc. Retrieving data graphs that contain a query graph is a key issue in these areas. This type of search is well-known as *subgraph search*. Formally, given a graph database $\mathcal{D} = \{g_1, g_2, \ldots, g_{|\mathcal{D}|}\}$ and 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. Figure 1 shows a running example of subgraph search, where a sample database composed of three data graphs 001, 002 and 003, are given in Figure 1(a) and a query graph q is given in Figure 1(b). The number beside each vertex is its id and the letter inside the vertex is its label. Graph 003 should be returned as a result since it contains q.



(a) A Sample Graph Database

(b) Query Graph

(c) Features

Figure 1: Running Example

Unfortunately, the subgraph search problem is hard in that it requires subgraph isomorphism checking of query q against each data graph g_i, which has proven to be NP-complete problem [1]. Indexing [2, 3, 4] is proposed to alleviate the overhead of pairwise isomorphism checks. In this approach, indexes are used to quickly filter out data graphs that are not possible in the result and produce candidate graphs. Then the candidate graphs are verified, i.e. whether the query graph is a subgraph of each candidate, by a subgraph isomorphism algorithm. The efficiency of this approach depends on the filtering power of each indexing methodology and how fast it produces candidate graphs. Other measures such as index construction time and update are also very important to the overall quality of the indexing method. Even with the existence of a high quality indexing method, efficient subgraph checking algorithm is very important since it is required to verify the candidates. Note that there are many scenarios in which lots of data graphs contain the query, and using any filtering process would return all these graphs as candidates to be finally verified.

EDBT/ICDT '13, March 18 - 22 2013, Genoa, Italy
Copyright 2013 ACM 978-1-4503-1597-5/13/03...$15.00.

Feature-based indexing [2, 3] is a well-known graph indexing mechanism that works by extracting subgraphs[1] of data graphs as indexing features, and using them to prune data graphs not having the features that are contained by the query. The major challenge facing this approach is that in order to have an index with good pruning quality, a multitude of database features have to be indexed. The higher the number of features to be indexed, the more possibility that they comprise a sufficiently discriminative feature set. Nevertheless, in addition to the big index size that results, the filtering time increases. This is because of the increase in the overhead of comparing query features with too many indexing features. Note that, a canonical representation is required to compare features. For general subgraphs, finding a unique representation is far from trivial.

Several feature-based indexing methods have been introduced in recent years. Many of them adopt data-mining techniques to effectively identify compact sets of features that possess higher pruning capabilities [3, 5, 6, 7, 8]. The main limitations of these methods are i) the decisions regarding the number, size, structure and pruning quality of the selected features crucially affect the cost of the mining process and consequently the index construction time, and ii) the quality of the selected features degrades over time due to frequent database updates which necessitates re-building the index from scratch. Other methods use exhaustive feature enumeration [2, 9, 10]. Though they support updating and do not require the costly mining process, they on the other hand are restricted to specific classes of features in order to keep the index size manageable. This of course has a negative impact on the pruning quality. The main objective of our research is to develop an indexing method having the salient aspects of all previous methods and avoiding their disadvantages.

In this paper, we propose an efficient indexing approach which improves the pruning power of current feature-based methods, neither by the costly feature selection nor by explicitly indexing a multitude of features. The idea at the heart of the method is based on the observation that the pruning quality of a group of features does not necessitate indexing all these features. Rather than using the expensive mining process to approximately capture this idea, we show, in this paper, that by indexing a simple feature structure combined with its degree sequence and neighborhood, we get a pruning quality as good as that of many of its supergraphs features. Thus, in our approach the pruning quality of multiple features is compressed into one feature with some neighborhood information encoded. Therefore, the overhead of indexing a multitude of features or selecting some of them is not required by our approach.

To use neighborhood for pruning, our method makes use of full information of each indexing feature, that is, the location/position of each occurrence of the feature in each data graph is remembered. We show that storing the mapping information of each feature's occurrences gives extra benefits such as: First, it improves the accuracy of feature containment in both query and data graphs; consequently, the pruning power is further improved. Second, the fact that neighborhood preserves local structural information surrounding occurrences allows to prune unmatched feature occurrences between the query and data graphs, thus cutting down the search space of subgraph matching, which significantly reduces the verification cost.

The index generated by our method is kept as small as possible by utilizing neighborhood sharing and vertex repetition. We further speed up occurrences matching by removing duplicated computations. Note that, a vertex may appear in different features and occurrences, and have neighborhood similar to that of other vertices. Moreover, real data graphs that come from the same application domain tend to share commonality, that is, a vertex with its neighborhood appears in many data graphs. We exploit these properties to minimize the index size and use caching to relieve the overhead of the repeated computations.

We implement the method by exhaustively enumerating simple paths of size up to *maxL*-edges as features ($maxL \leq 3$ in experiments). The index generated is called PathIndex. A novel path-at-a-time verification method that benefits from the indexed path occurrences, and the occurrences pruning method is introduced. The salient aspect of this method is that data graphs are not required, indexed occurrences are used instead. Extensive experiments show that PathIndex's pruning is highly efficient, its construction is very fast, and has smaller size compared to other indexes. Further experiments show that PathIndex based subgraph search outperforms that of the state-of-the-art indexing methods on both real and synthetic datasets.

2. PRELIMINARIES

Given two sets of labels, Σ_1 and Σ_2, a labeled graph g is defined as a triple (V_g, E_g, l), where $V_g = \{v_1, v_2, \ldots, v_{|V|}\}$ is the set of vertices, $E_g \subseteq V_g \times V_g$ is the set of edges (directed or undirected), and l is a labeling function: $V_g \rightarrow \Sigma_1$ and $E_g \rightarrow \Sigma_2$. The size of g is denoted by $|g| = |E_g|$. This paper focuses on simple, undirected graphs with vertex and edge labels. However, for presentation ease, we consider labeled nodes only.

A graph $g = (V_g, E_g, l)$ is a *subgraph* of another graph $g' = (V_{g'}, E_{g'}, l')$ (or g' is a *supergraph* of g), denoted $g \subseteq g'$, if there exists a *subgraph isomorphism* from g to g'. We may simply say that g' contains g.

Definition 1. **Subgraph isomorphism.** *A subgraph isomorphism is an* injective *function* $O : V_g \rightarrow V_{g'}$*, such that (1)* $\forall v \in V_g, l(v) = l'(O(v))$. *(2)* $\forall (u, v) \in E_g, (O(u), O(v)) \in E_{g'}$*, and* $l(u, v) = l'(O(u), O(v))$.

According to Def. 1, the mapping information $O(g, g') = \{O(v_i) : v_i \in V_g\}$ identifies a subgraph of g' called an *occurrence* of g in g'. The graph g' may contain many occurrences of the graph g. Two occurrences $O_1(g, g')$ and $O_2(g, g')$ are considered *redundant* if their corresponding subgraphs are automorphic. A *graph automorphism* is an isomorphism from the graph to itself. The concept of *graph isomorphism* can be defined analogously as Def. 1 by using a bijection instead of an injection.

Definition 2. **Labeled Path.** *A* path $u \rightsquigarrow u'$ *from a vertex* u *to a vertex* u' *in a graph* $g = (V_g, E_g, l)$ *is a sequence* $v_0, v_1, \ldots, v_k$ *of vertices such that* $u = v_0$ *and* $u' = v_k$*, and* $(v_{i-1}, v_i) \in E_g$. *A path is* simple *if all of its vertices are distinct. If the vertex label is used instead of its id, for each vertex in the path, the path is called* labeled *path.*

[1] Path, tree, or general subgraphs.

Definition 3. **Induced Subgraph.** *Given a graph $g = (V_g, E_g, l)$. For each $U \subseteq V_g$, the induced subgraph of g defined on the vertex subset U, denoted as $I_g(U) = (U, E')$, is the graph composed of all the g's edges that connect pairs of vertices of U, that is, $E' = (U \times U) \cap E_g$.*

Subgraph Search Problem. Given a graph database $\mathcal{D} = \{g_1, g_2, \ldots, g_{|\mathcal{D}|}\}$ and a query graph q. The problem of subgraph search is to find from $\mathcal{D}$ a set of graphs which contain q, $\mathcal{D}_q$, i.e., $\mathcal{D}_q = \{g : g \in \mathcal{D} \wedge q \subseteq g\}$.

Subgraph query processing by pairwise graph comparison is very expensive due to data set size and computation hardness of subgraph test (subgraph isomorphism is NP-Complete[1]). Reducing the number of graph comparisons through the framework of filtering-and-verification, i.e. Indexing, is the adopted practical solution. Next, we introduce feature-based indexing since it is the basis of our study. From now on, a data graph, query graph and feature graph are called a graph, query and feature, respectively.

Algorithm 1: Query_Processing($q, I, \mathcal{D}$)

Input: q: query graph; I: feature-based index; $\mathcal{D}$: database;
Output: $\mathcal{D}_q$ is a set of matched graphs, initialized empty;

```
1: $\mathcal{F}_q = \{f : f \subseteq q \wedge f \in I\}$;
2: $C_q = \bigcap_{f \in \mathcal{F}_q} \mathcal{D}_f$;
3: for each $g \in C_q$ do
4:     if $q \subseteq g$ then
5:         $\mathcal{D}_q = \mathcal{D}_q \cup \{g\}$
6: return $\mathcal{D}_q$
```

2.1 Feature-based Index

A feature-based index $I = \{(f_i, \mathcal{D}_{f_i}) : f_i \in \mathcal{F}\}$ is a set of indexed items. Here $\mathcal{F}$ is a set of features, where f_i is a subgraph in the database which can be a path, a tree, or a general subgraph. And $\mathcal{D}_{f_i}$ is a set of graph ids that contain f_i, i.e., $\mathcal{D}_{f_i} = \{g \in \mathcal{D} : f_i \subseteq g\}$. The filter-and-verification framework using I is outlined in Algorithm 1. As shown, the filtering phase and the verification phase are specified in lines 1-2 and lines 3-5, resp. Line 1 retrieves the indexing features contained by the query q. Line 2 gets graphs that contain all the features appearing in q, which is known as the candidate set C_q. Lines 3-5 perform subgraph isomorphism test for each candidate graph g. If there is a subgraph isomorphism from q to g, g is added to the answer set $\mathcal{D}_q$, i.e., the set of matched graphs.

As an example, suppose the two features f_1 and f_2 in Figure 1(b) are indexed, i.e., $I = \{(f_1, D_{f_1}), (f_2, D_{f_2})\}$. By following Algorithm 1, the graphs 001, 002 and 003 will be returned as candidates since they contain the features f_1 and f_2 which are contained by the query. Thus, the verification will be performed on the whole example database even though the graph 003 is the only graph that contains the query.

The above example highlights the poor pruning of current setting of feature-based filtering scheme. To enhance pruning of this scheme, a large number of features have to be indexed, resulting in a large index with increasing filtering time; or alternatively, much more time would have been needed offline to select a compact set of discriminative features. Moreover, since the index as defined contains no useful information for speeding up the verification process, in scenarios where all graphs, or most of them, contain the query, extra verification cost would be added to the overall query processing. Next, we show that, by encoding feature occurrences, their degree sequences and neighborhoods, f_1 and f_2 could be used to reduce not only the candidate set size but also the search space of subgraph matching on the remaining graphs.

3. NEW FEATURE-BASED PRUNING APPROACH

Given a graph $g = (V_g, E_g)$. For each vertex $v \in V_g$, define the *neighborhood* of v in g, denoted $N_g(v)$, to be the set of vertices u in g such that there exists an edge in g connecting u and v. That is, $N_g(v) = \{u \in V_g : (u, v) \in E_g\}$. Let $deg_g(v)$ denote the degree of v in g. For graphs g and g', $g \subseteq g'$, we define the degree sequence of an occurrence of g in g' as follows.

Definition 4. **Degree Sequence of an Occurrence.** *Let $O(g, g') = \{O(v_i) : v_i \in V_g\}$ be an occurrence of g in g'. The* degree sequence *of $O(g, g')$, denoted $deg_O(g, g')$, is a sequence of integers $<d_1, d_2, \ldots, d_{|V_g|}>$ of length $|V_g|$, where each $d_i = deg_{g'}(O(v_i))$, $\forall v_i \in V_g$.*

As an example, in Figure 1, the query q contains one occurrence of the feature f_1, given as $O(f_1, q) = \{v_5, v_4, v_2\}$, and its degree sequence is $deg_O(f_1, q) =$<2,3,4>. The graph 002 also contains one occurrence of f_1 which is $O(f_1, 002) = \{v_6, v_5, v_7\}$; its degree sequence is $deg_O(f_1, 002) =$<2,5,4>.

Given graphs g, g' and g''. Let g' and g'' contain k and m distinct occurrences of g, resp. Based on degree sequence of occurrences, we define compatible occurrences as follows.

Definition 5. **Compatible Occurrences.** *Given two occurrences $O_i(g, g')$ and $O_j(g, g'')$ of g in g' and g'', respectively. Let $deg_{O_i}(g, g') = <d'_1, d'_2, \ldots, d'_{|V_g|}>$ and $deg_{O_j}(g, g'') = <d''_1, d''_2, \ldots, d''_{|V_g|}>$ be their degree sequences. $O_j(g, g'')$ is said to be* compatible *to $O_i(g, g')$ iff $d'_l \leq d''_l$, $1 \leq l \leq |V_g|$.*

Based on Def. 5, it follows that $O(f_1, 002)$ is compatible to $O(f_1, q)$. The main problem with Def. 5 is that in order to decide the compatibility between two occurrences, we may end up comparing the degree sequence of all redundancies of both occurrences. For example, given the redundancy $O'(f_1, 002) = \{v_7, v_5, v_6\}$ of $O(f_1, 002) = \{v_6, v_5, v_7\}$. If we only consider $O'(f_1, 002)$ instead of $O(f_1, 002)$ during degree comparisons, we conclude that $O(f_1, q)$ has no compatible occurrence in the graph 002. Instead, we have to compare every pair of redundancies of both occurrences. Below, we show that one specified redundancy of each occurrence is sufficient for degree sequence comparison. First, we show that the main cause of redundancy is the existence of isomorphism relationship among graph vertices.

Definition 6. **Isomorphic Vertices.** *Given a graph $g = (V_g, E_g, l)$, the two vertices $v_i, v_j \in V_g$ are isomorphic iff for each vertex $u \in V_g$, $u \neq v_i$ there exists a vertex $u' \in V_g$, $u' \neq v_j$ such that the labeled shortest paths $v_i \rightsquigarrow u$ and $v_j \rightsquigarrow u'$ are isomorphic.*

It is straightforward to show that the isomorphism relationship between graph vertices is an equivalence relation. Therefore, V_g can be divided into equivalent classes of isomorphic vertices. Let $IsoSet_i$ denote one such class. Obviously, there are at most $\prod_i(|IsoSet_i|)!$ redundancies of each occurrence of g in any graph. As an example, the feature f_1 has two classes of isomorphic vertices $IsoSet_1 = \{v_1, v_3\}$ and $IsoSet_2 = \{v_2\}$. Thus, there are $2! \times 1! = 2$ redundancies of each occurrence of f_1 in any graph. Hereafter, we consider the redundancy that maps the isomorphic vertices to the ordered ones, where the ordering is taken based on vertex degree. We call this redundancy *canonical*. For example, we will consider $O(f_1, 002)$ not $O'(f_1, 002)$ as canonical, since the two isomorphic vertices v_1 and v_3 of f_1 are mapped under O to the vertices v_6 and v_7, respectively, and $deg_{002}(v_6) \le deg_{002}(v_7)$. Lemma 1 shows that the compatibility test on canonical occurrences is sufficient.

Lemma 1. *Let $O_i(g, g')$ and $O_j(g, g'')$ be two canonical occurrences of g in g' and g'', resp. If O_j is not compatible to O_i then no one of O_j redundancies is compatible to any redundancy of O_i.*

proof: Let $deg_{O_i}(g, g') = <d'_1, d'_2, \ldots, d'_{|V_g|}>$ and $deg_{O_j}(g, g'') =<d''_1, d''_2, \ldots, d''_{|V_g|}>$ be O_i and O_j degree sequences, resp. Suppose O_j is not compatible to O_i, then there exists l such that $d'_l > d''_l$. Any redundancy of O_j is obtained by exchanging the mapping at two isomorphic vertices of g. Suppose there exists $v_k \in V_g$, $k > l$, such that v_k is isomorphic to v_l and $d''_k > d'_l$. By exchanging $O_j(v_k)$ with $O_j(v_l)$, we obtain a new redundancy that makes $d'_l < d''_l$, however, at position k, we still have $d'_k > d''_k$, that means the new obtained redundancy of O_j is not compatible to the canonical O_i. Analogically, we can show that the canonical O_j is not compatible to any redundancy of O_i. Let $k < l$. By exchanging $O_i(v_k)$ with $O_i(v_l)$, we obtain a new redundancy of O_i that makes $d'_l < d''_l$, however, at position k, we still have $d'_k > d''_k$, that means the canonical O_j is not compatible to the new obtained redundancy of O_i. ■

Given the k and m distinct occurrences of g in g' and g'', and their compatibilities, we construct a bipartite graph of g's occurrences called *Occurrences Bipartite Graph*, denoted $B_g(g', g'')$ and defined as follows.

Definition 7. **Occurrences Bipartite Graph.** *Occurrences Bipartite Graph of g w.r.t. graphs g' and g'' is the graph $B_g(g', g'') = (V_1, V_2, E)$, where $V_1 = \{O_i(g, g')\}_{i=1}^{k}$ and $V_2 = \{O_j(g, g'')\}_{j=1}^{m}$, the vertex sets of B_g, are the k and m distinct occurrences of g in g' and g'', resp. For any two occurrences $u \in V_1$ and $v \in V_2$, if u is compatible to v, then $(u, v) \in E$ is an edge of B_g, called bipartite edge.*

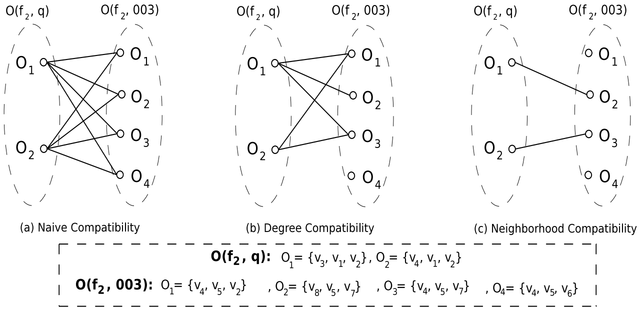
Definition 8. **Occurrence Matching Candidates.** *Given $B_g(g', g'') = (V_1, V_2, E)$. For each occurrence $u \in V_1$, $C(u) = \{v \in V_2 : (u, v) \in E\}$ is the set of matching candidates of u in V_2.*

A *matching* M of a bipartite graph $G = (V_1, V_2, E)$ is a subset of the bipartite edges with the property that no two edges of M share the same vertex; M is called *semi-perfect* if every vertex in V_1 is matched. Theorem 1 makes use of occurrences bipartite graph of each feature contained by the query and graphs for effective pruning.

Theorem 1. *Given three graphs f, q and g s.t. $f \subseteq q$ and $f \subseteq g$. If $q \subseteq g$, then $B_f(q, g)$ has a semi-perfect matching.*

Thus, according to Theorem 1, if there exists an indexing feature f such that $f \subseteq q$, $f \subseteq g$, and $B_f(q, g)$ does not contain a semi perfect matching, then q is not subgraph isomorphic to g, and g can be safely pruned. Since the number of bipartite edges (matching candidates) plays a dominant role in the complexity of any bipartite matching (or a subgraph matching) algorithm, so minimizing these edges (candidates) is crucial for the overall performance of subgraph search. Figure 2 shows $B_{f_2}(q, 003)$, where matching occurrences is based on naive (Figure 2(a)) and degree sequences (Figure 2(b)) compatibilities. Notice that the number of matching candidates of f_2 occurrences in q is reduced from 8 to 5 candidates in the graph 003 under the degree sequences compatibility.

In the next subsection, we strengthen the compatibility definition in order to minimize the number of matching candidates of each feature occurrence. Corollary 1 helps in pruning some unpromising graphs without carrying out the bipartite matching test.


O(f2, q)
O(f2, 003)
O1
O2
O1
O2
O3
O4
O(f2, q)
O(f2, 003)
O1
O2
O1
O2
O3
O4
O(f2, q)
O(f2, 003)
O1
O2
O1
O2
O3
O4
(a) Naive Compatibility

(b) Degree Compatibility

(c) Neighborhood Compatibility

O(f2, q): O1= {v3, v1, v2}, O2= {v4, v1, v2}
O(f2, 003): O1= {v4, v5, v2} , O2= {v8, v5, v7} , O3= {v4, v5, v7} , O4= {v4, v5, v6}

Figure 2: $B_{f_2}(q, 003)$

Corollary 1. *Let $\{O_i(f, q)\}_{i=1}^{k}$ and $\{O_j(f, g)\}_{j=1}^{m}$ be the k and m distinct occurrences of f in q and g, resp. If $k > m$, or there exists an occurrence $O_i(f, q)$ such that it has no matching candidates in $\{O_j(f, g)\}$, then q is not subgraph isomorphic to g.*

Whereas the first condition of Corollary 1 is helpful in early pruning of graphs, the second condition is effective in pruning a graph g even before constructing $B_f(q, g)$. As an example, based on the degree sequence compatibility, the two occurrences of f_2 in q have no matching occurrences of f_2 in the graph 001. Therefore, according to the second condition of Corollary 1, the graph 001 does not contain q. Note that the first condition of Corollary 1 would return the graph 001 as a candidate. For the remaining two graphs 002 and 003, a semi-perfect matching exists in f_1 and f_2 bipartite graphs. Then, they may contain the query. Next, we show that by exploiting local structures surrounding feature occurrences, more graphs can be filtered.

3.1 Occurrences Neighborhood

For each occurrence of a feature f in the query q, there must exist at least one distinct corresponding occurrence of f in g in order for g to be qualified as a candidate. In fact, occurrences compatibility based on degree sequences is weak. Local structures should also be preserved around corresponding occurrences. In Figure 2 (b), $O_1(f_2, 003)$ is compatible to $O_1(f_2, q)$ based on their degree sequences. However, if we look at the connections of vertex $v_2 \in O_1(f_2, q)$ and its corresponding $v_2 \in O_1(f_2, 003)$, we find that $v_2 \in O_1(f_2, q)$ is connected to vertices with the labels A, A, A, B. On the other hand, its corresponding $v_2 \in O_1(f_2, 003)$ is connected to vertices with different labels A, A, C, C; consequently, $O_1(f_2, 003)$ is not compatible to $O_1(f_2, q)$. Below, we strengthen the correspondence definition of occurrences by taking into account local structures around occurrences. To capture these structures, we start by defining *relative neighborhood* of a vertex and then its *Neighborhood Graph*.

Definition 9. **Vertex Relative Neighborhood.** *Given two graphs f and g, and let $O(f, g)$ be an occurrence of f in g. For each vertex $u \in O(f, g)$, there exists $v_i \in V_f$ such that $u = O(v_i)$. The neighborhood of u in g relative to f, called u's relative neighborhood, denoted $N_{O(f,g)}(u)$, is given by $N_{O(f,g)}(u) = N_g(u) \backslash \{O(v_j) : v_j \in N_f(v_i)\}$.*

For example, given the occurrence $O(f_1, 002) = \{v_6, v_5, v_7\}$ then $N_{O(f_1,002)}(v_6) = \phi$, $N_{O(f_1,002)}(v_5) = \{v_8, v_4, v_2\}$ and $N_{O(f_1,002)}(v_7) = \{v_8, v_4\}$. Let $N^l_{O(f,g)}(u)$ denote the relative neighborhood of u by taking the vertex label instead of vertex id. It follows, $N^l_{O(f_1,002)}(v_5) = \{A, A, B\}$ and $N^l_{O(f_1,002)}(v_7) = \{A, A\}$. Based on Def. 9, given $X \subseteq O(f, g)$, we define the relative neighborhood of X, denoted $N^l_{O(f,g)}X$, to be the union of all relative neighborhoods of X's vertices, that is, $N^l_{O(f,g)}X = \bigcup_{u \in X} N^l_{O(f,g)}(u)$. Thus, $N^l_{O(f_1,002)}\{v_6, v_7\} = \phi \cup \{A, A\} = \{A, A\}$.

Definition 10. **Vertex Neighborhood Graph.** *For each vertex $u \in O(f, g)$. Define Neighborhood Graph of u, denoted $NG_{O(f,g)}(u)$, to be the induced graph $I_g(N^+_{O(f,g)}(u))$ after removing the following edges $\{(u, v) : v \notin O(f, g)\}$, where $N^+_{O(f,g)}(u) = N_{O(f,g)}(u) \cup \{u\}$.*

For example, $NG_{O(f_1,q)}(v_2) = (\{v_1, v_3, v_2\}, \{(v_1, v_3)\})$. If we consider the vertex label instead of its id, $NG^l_{O(f_1,q)}(v_2) = (\{A, A, B\}, \{(A, A)\})$. As above, we define the neighborhood graph for $X \subseteq O(f, g)$, denoted $NG^l_{O(f,g)}X$, to be the union of all neighborhood graphs of X's vertices, that is, $NG^l_{O(f,g)}X = \bigcup_{u \in X} NG^l_{O(f,g)}(u)$, e.g., $NG^l_{O(f_1,q)}\{v_5, v_2\} = (\{A, A, B, B\}, \{(A, A)\})$ and $NG^l_{O(f_1,002)}\{v_6, v_7\} = (\{A, A, B, B\}, \phi)$.

Definition 11. **Compatible Occurrences (revisited)** *Given two occurrences $O_i(f, q)$ and $O_j(f, g)$ of f in q and g, resp. Let $\{IsoSet_h\}^n_{h=1}$, $n \leq |V_f|$, be the multiset of isomorphic classes of f. $O_j(f, g)$ is said to be* compatible *to $O_i(f, q)$ iff $NG^l_{O_i(f,q)}O_i(IsoSet_h) \subseteq NG^l_{O_j(f,g)}O_j(IsoSet_h)$, $\forall h$, where $O(IsoSet_h) = \{O(v) : v \in IsoSet_h\}$.*

Corollary 2. *Given three graphs f, q and g. Let $B_f(q, g)$ be the bipartite graph based on the compatibility Def. 11. If $q \subseteq g$ then $B_f(q, g)$ has a semi-perfect matching.*

The compatibility condition of Def. 11 comprises two inclusion tests on every corresponding classes of vertices: one on the vertex sets and the other on the edge sets of the corresponding neighborhood graphs. Generally, it is flexible to utilize both or any of the two inclusion tests depending on which parts of the neighborhood graphs are indexed. Moreover, the test on vertex sets can be safely replaced by a test on (relative) neighborhoods of corresponding classes of vertices. This flexibility allows us to control the index more efficiently (see next section). Consider, for example, applying this new compatibility condition on the remaining graphs 002 and 003. For the graph 002, the two corresponding neighborhood graphs $NG^l_{O(f_1,q)}\{v_5, v_2\}$ and $NG^l_{O(f_1,002)}\{v_6, v_7\}$ given above have similar vertex sets, but the first graph has the edge (A, A) while the second contains no edges. It implies that $O(f_1, 002)$ is not compatible to $O(f_1, q)$. Since $O(f_1, 002)$ is the unique occurrence of f_1 in 002, this graph can be safely pruned. For the graph 003, even though it passes the semi-perfect matching test for f_1 and f_2 bipartite graphs, the minimal number of matching candidates obtained for each feature occurrence will speed up the verification process on this graph (see Section 5.1). Figure 2(c) shows that $B_{f_2}(q, 003)$ has a minimal number of bipartite edges (matching candidates) after applying the compatibility condition of Def. 11.

The benefit of indexing neighborhood graphs of a given feature occurrence is justified by the following theorem.

Theorem 2. *Given $O(f, g)$, an occurrence of f in g. Indexing $O(f, g)$ and $\{NG^l_{O(f,g)}(u) : u \in O(f, g)\}$, the neighborhood graphs of $O(f, g)$, is equivalent to indexing $O(f, g)$ and all its supergraphs up to the graph $NG^l_{O(f,g)}O(f, g)$.*

Theorem 2 tells us that, indexing an easy-to-obtain, simple feature structure with its neighborhood compresses many of its supergraph features which are indexed in the traditional feature-based methods through the expensive mining process. Section 5 gives an example, where small paths are used as features.

4. INDEXING FRAMEWORK

The feature-based indexing scheme given in Section 2.1 is feature oriented, that is, each feature is associated with an inverted list of graphs ids that contain it. Here, we adopt an alternative scheme called *graph oriented*. In this scheme, each graph has a list of features ids that are contained by the graph, and additionally each feature id is attached with feature occurrences and their neighborhood information in the graph. Formally, given a graph database $\mathcal{D} = \{g_1, g_2, \ldots, g_{|\mathcal{D}|}\}$ and a set of features $\mathcal{F} = \{f_1, \ldots, f_{|\mathcal{F}|}\}$. Let $\{O_i(f, g)\}$ be the set of all occurrences of a feature f in a graph g. Our index is given by $I = \{(g, \mathcal{F}'_g) : g \in \mathcal{D}\}$, where $\mathcal{F}'_g$ is the set of features that are contained by g and associated with their occurrence matrices, that is, $\mathcal{F}'_g = \{< f, \mathcal{O}(f, g) >: f \in \mathcal{F} \wedge f \subseteq g\}$. The *Occurrence Matrix* of f in g, denoted $\mathcal{O}(f, g)$, is a size $|\{O_i(f, g)\}| \times |V_f|$ matrix, where each row represents an occurrence of f in g, and each matrix entry is attached with a two-lists container to maintain the vertex labels and edges of the corresponding neighborhood graph. For instance, in graph 003 we index two occurrences of f_1 and four occurrences of f_2. Figure 3(a) spots on the two-lists container attached with each vertex in the occurrence $O(f_1, 002) = \{v_6, v_5, v_7\}$.

Though the index seems to be large as we maintain the location and two neighborhood lists for each vertex in each feature occurrence, the adopted graph oriented scheme allows us to answer queries while the major portion of the index remains on disk. Moreover, properties such as neighborhood sharing and vertex repetition can be used to keep the index size minimized, as we see next.

L/id	v_6	v_5	v_7
$V_{NG^l_O(v)}$	{B}	{A,A,A,B}	{A,A,B}
$E_{NG^l_O(v)}$	ϕ	{(A,B)}	ϕ

(a)

f/g	001	002	003
f_1	1	1	2
f_2	3	3	4

(b)

Figure 3: (a) The two-lists containers attached with $O(f_1, 002) = \{v_6, v_5, v_7\}$, (b) Feature Graph Matrix

4.1 Filtering

In this subsection, we present the basic filtering algorithm followed by an analysis of its complexity. Next, we propose efficient optimizations. Algorithm 2 outlines the filtering framework. Given a query q, $\mathcal{F}'_q$ is constructed. The index I is kept on disk, and used to filter false graphs as many as possible based on Corollary 2 as follows. For each graph g, $\mathcal{F}'_g$ is accessed from I, and used with $\mathcal{F}'_q$ to construct $B_f(q, g)$ for each feature f that occurs in q and g. If $B_f(q, g)$ does not have a semi-perfect matching for any f, then g is pruned.

```
Algorithm 2: Basic_Filtering(q, I)
Input: q: the query; I = {(g, F'_g) : g ∈ D}, the graph index;
Output: C_q: a set of candidate graphs, initially C_q = D;
1: $\mathcal{F}'_q = \{< f, \mathcal{O}(f, q) >: f \in I \wedge f \subseteq q\}$;
2: for each $g \in C_q$ do
3:     Load $\mathcal{F}'_g$;
4:     for each $f \in \mathcal{F}'_q \wedge f \in \mathcal{F}'_g$ do
5:         Process $\mathcal{O}(f, q)$ and $\mathcal{O}(f, g)$ to construct $B_f(q, g)$;
6:         if $B_f(q, g)$ has no semi-perfect matching then
7:             $C_q = C_q \setminus \{g\}$;
8:             break;
9: return $C_q$
```

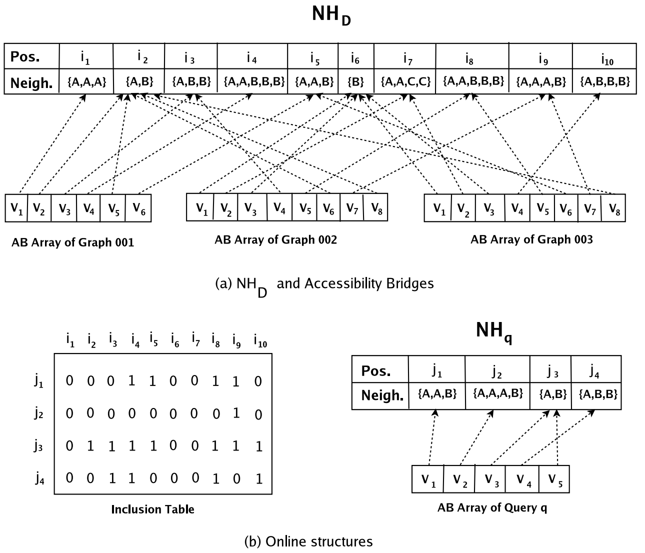
Suppose that k and m are the average number of occurrences taken over all features f in q and graphs g, resp. Line 5 of Algorithm 2 constructs $B_f(q, g) = (b_{ij})$, the bipartite matrix between occurrences of f in q and g, i.e., it is $k \times m$ matrix of 0 and 1 entries, where $b_{ij} = 1$, if the occurrences $O_i(f, q)$ and $O_j(f, g)$ are compatible and $b_{ij} = 0$, otherwise. Each entry b_{ij} is computed in $Iso \times c$ amortized time, where Iso is the average number of isomorphic classes taken over all features f, and c is the average cost of the neighborhood and edge inclusion tests of corresponding classes. For each feature f, the bipartite matrix is then constructed in $(Iso \times c \times k \times m)$ amortized time. Let c' be the cost that semi-perfect matching test (line 6) can take, and since $k \leq m$, then the worst-case complexity of lines 5-6 is given by $O(Iso \times c \times m^2 + c')$. Thus, the worst-case time complexity of the filtering algorithm is dominated by the complexity of the nested for loop at lines 2-8, which is $O((Iso \times c \times m^2 + c')|\mathcal{F}'_q||C_q|)$.

4.2 Optimizations

Our approach improves the pruning power significantly but the filtering time needs to be improved. To alleviate the filtering overhead, we consider the following three optimizations:

1. Minimizing the initial candidate set C_q.
2. Minimizing the inclusion tests $Iso \times c \times m^2$.
3. Minimizing $|\mathcal{F}'_q|$.

Objective 1 is easy to achieve by adding to the index a memory-resident component called *feature-graph matrix* [11], denoted by $M^{\mathcal{F}}_{\mathcal{D}}$. Figure 3(b) shows $M^{\mathcal{F}}_{\mathcal{D}}$ for the example database given in Figure 1(a). Each column of $M^{\mathcal{F}}_{\mathcal{D}}$ corresponds to a target graph, whereas each row corresponds to a feature being indexed. Each entry records the number of distinct occurrences of a specific feature in a target graph. With this component, we can apply the first condition of Corollary 1 as a preliminary filter to reduce the initial candidates C_q. Since we enumerate every feature occurrence, $M^{\mathcal{F}}_{\mathcal{D}}$ is constructed with no time as a by-product of the enumeration process. The main problem with $M^{\mathcal{F}}_{\mathcal{D}}$ is when the number of indexing features is large such that much memory is needed for it. To resolve this, the list $M^{\mathcal{F}}_g = \{< f, |\mathcal{O}(f, g)| >: f \in \mathcal{F}'_g\}$ is maintained on disk for each graph g and accessed while processing g.



NH$_D$

Pos.	i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8	i_9	i_{10}
Neigh.	{A,A,A}	{A,B}	{A,B,B}	{A,A,B,B,B}	{A,A,B}	{B}	{A,A,C,C}	{A,A,B,B,B}	{A,A,A,B}	{A,B,B,B}

v_1 v_2 v_3 v_4 v_5 v_6
AB Array of Graph 001
v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8
AB Array of Graph 002
v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8
AB Array of Graph 003

(a) NH$_D$ and Accessibility Bridges

	i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8	i_9	i_{10}
j_1	0	0	0	1	1	0	0	1	1	0
j_2	0	0	0	0	0	0	0	0	1	0
j_3	0	1	1	1	1	0	0	1	1	1
j_4	0	0	1	1	0	0	0	1	0	1

Inclusion Table

NH$_q$

Pos.	j_1	j_2	j_3	j_4
Neigh.	{A,A,B}	{A,A,A,B}	{A,B}	{A,B,B}

v_1 v_2 v_3 v_4 v_5
AB Array of Query q

(b) Online structures

Figure 4: (a) NH$_D$ and AB arrays of the example database (b) NH$_q$ and the inclusion table

To achieve objective 2, we observe that many vertices in the database share the same (relative) neighborhood. Moreover, a vertex may appear in many features of the data graph. Inspired by this observation, we build a memory-resident component called *data Neighborhood Matrix*, denoted NH$_{\mathcal{D}}$, a matrix to maintain distinct (relative) neighborhoods in the database $\mathcal{D}$. Thus, for each entry in the occurrence matrix of a given feature f in a graph g, we replace the vertex (relative) neighborhood list by a pointer to its position at NH$_{\mathcal{D}}$. In the online process, given a query q, the query neighborhood matrix NH$_q$ is also constructed. Then, a binary lookup table, called *inclusion table*, is created to maintain the inclusion relationship between distinct neighborhoods in NH$_q$ and in NH$_{\mathcal{D}}$. The inclusion table relieves the computation overhead caused by the repeated neighborhood compatibility checks of occurrences. In fact, by using the inclusion table, the complexity term $Iso \times c \times m^2$ is reduced to $Iso \times m^2 + c''$, where c'' is the cost of building the inclusion table. Furthermore, using NH$_{\mathcal{D}}$ significantly reduces the index size as well. Note that, a pointer is used in place of the whole (relative) neighborhood list. Moreover, if the vertex neighborhood is indexed instead of its relative neighborhood, the index size can be further improved by removing all neighborhood pointers and using, instead, an array of pointers of size $|V_g|$ with each graph g, called *Accessibility Bridge (AB)* array. In the *AB* array, each slot u holds a pointer to the vertex u neighborhood list at $\mathrm{NH}_{\mathcal{D}}$. Note also that, computing neighborhoods is easier than computing relative neighborhoods. Figure 4 (a) shows the data Neighborhood Matrix NH_D and the Accessibility Bridges of the example database. These structures replace the two-lists containers associated with features occurrences in the index. Figure 4 (b) shows the inclusion table and NH_q constructed online. Likewise, a similar approach can be employed on distinct edge lists of neighborhood graphs, if those edges are indexed.

Objectives 3 is not so easy since it depends on how features are selected. However, as suggested in [3], maximal features in $\mathcal{F}'_q$ are sufficient. In our setting, a feature is *maximal* if there exists an occurrence of it that is not contained by any other feature. Considering maximal features saves repeated computations performed on their subgraphs features.

5. IMPLEMENTATION

Previously, we presented an approach that improves the filtering power of any group of features regardless of feature type. To show how this approach works in practice, we implemented it using paths as features; the index generated is called PathIndex. *Why paths?* A path with encoded neighborhood information compresses many of its supergraphs (Theorem 2). Moreover, paths are easy to manipulate, as problems like canonicalization are much easier to handle. It also helps to keep the index size reasonable.

PathIndex exhaustively enumerates simple paths of size up to *maxL*-edges as features. Structures such as $\mathrm{NH}_{\mathcal{D}}$ and $\mathrm{M}^{\mathcal{F}}_{\mathcal{D}}$ are constructed in PathIndex. Also, neighborhoods are indexed instead of relative neighborhoods, and the edge lists of neighborhood graphs are not considered. In PathIndex, every path is encoded, and since the same path may appear in different graphs, we build a path code dictionary to store all distinct path codes. Note that, path code dictionary is easy for update maintenance. We only need to compute the path codes for the inserted (or deleted) graphs, and then insert them into (or delete them from) path code dictionary.

5.1 Novel Verification Approach

The verification step checks every graph survived the filtering process by an exact subgraph isomorphism test. Traditional graph matching methods perform subgraph test in a vertex-at-a-time manner. The basic verifier Ullman [12], for example, explores a tree-structured search space considering all possible vertex-to-vertex correspondences from the query to the graph. Since, in our framework, after filtering and according to occurrences pruning method, every path occurrence in the query has a compact set of compatible occurrences in every candidate graph (Section 3), hence the query could be answered by considering it in a path-at-a-time manner instead of vertex-at-a-time. Each compatible occurrence is, in fact, a local match to its corresponding query path occurrence. If the query is subgraph isomorphic to the graph, then some of these local matches could be combined together to produce a global match to the query. Note, however, that the query might include several paths. To effectively use the path-at-a-time scenario, we have to only consider the query disjoint paths.

Definition 12. **Disjoint paths.**
Distinct paths in a graph g are called disjoint *if they are edge disjoint, but not necessarily node disjoint.*

Algorithm 3 presents our path-at-a-time verifier. While query processing, the query q is decomposed into a set of disjoint paths $DP(q)$. Each path $p \in DP(q)$ is associated with a set of compatible paths $C(p)$ in the graph g. Some of these compatible paths will be used jointly to match the query as follows. The algorithm uses a vector $M = \{m_1, m_2, \ldots, m_{|V_q|}\}$ to denote which vertices of q have been mapped at an intermediate state of the computation. Here, $m_i \neq 0$ indicates that the ith vertex of q has been mapped. Another vector $H = \{h_1, h_2, \ldots, h_{|V_g|}\}$ is used to record the reverse mapping from g to q. Here, $h_i = j$ indicates that the jth vertex of q has been mapped to the ith vertex of g. The core procedure *Recursive_Search* uses local matches p' from $C(p_i)$ (lines 6-7) and proceeds step by step by recursively adding the subsequent local matches from $C(p_{i+1})$ (line 8), or outputs matched if every vertex of q has a counterpart in g (line 9). If p' exhausts all local matches in $C(p_i)$ and still cannot find a feasible matching, Recursive_Search backtracks to the previous state for further exploration (line 10). Function *Joinable* (lines 11-14) examines the feasibility of adding the current local match p' to join the previous ones. If that match conflicts with a previously chosen one, then we consider another local match from $C(p_i)$.

Algorithm 3: `Verify`$(DP(q), \mathcal{C}, M, H)$

Input: $DP(q) = \{p_1, \ldots, p_{|DP(q)|}\}$ disjoint paths covering q;
$\mathcal{C} = \{C(p_1), \ldots, C(p_{|DP(q)|})\}$, $C(p_i)$ is the set of compatible paths to p_i in g;
M: the mapping vector, initialized by all 0;
H: the reverse mapping vector initialized by all 0;
Output: *matched*: $q \subseteq g$, *unmatched*: otherwise;

```
1:  if Recursive_Search($p_1$) then return matched;
2:  return unmatched;
3:Procedure Recursive_Search($p_i$)
4:  for each $p' \in C(p_i)$ do
5:     if not Joinable($p', M$) then continue;
6:     for each $v \in V_{p'}$ do
7:        $m_u = v$; $h_v = u$; /*$u$ is mapped to $v$ in local match*/
8:     if $i < |DP(q)|$ then Recursive_search($p_{i+1}$);
9:     else if $m_u \neq 0, \forall u$ then return matched;
10:    for each $u \in (V_{p_i} \setminus \bigcup_{j<i} V_{p_j})$ do $m_u = 0$;
11:Function boolean Joinable($p', M$)
12:  for each $v \in V_{p'}$ do /*$u$ is mapped to $v$ in local match*/
13:     if $(m_u \neq 0 \wedge m_u \neq v)$ || $h_v \neq 0$ then return FALSE;
14:  return TRUE;
```

The product $\prod_{i=1}^{|DP(q)|} C(p_i)$ forms the total search space of Algorithm 3. To effectively use this algorithm, query paths should be chosen such that $|DP(q)|$ and $|C(p_i)|$ are minimized. For a given graph, there are multiple disjoint path decompositions. Algorithm 4 finds a compact set of disjoint paths that cover q. The algorithm works as follows. Given the set of all limited-size, simple paths P_q generated from the query q. P_q is processed in descending order of path size. For each encountered path $p \in P_q$, we check if removing p from the query disconnects it or not. If so, i.e., the resulting graph is disconnected, p is not considered and the search continue for another one. If, on the other hand, the resulting graph still connected , p is selected to be in the cover and removed from the query. Theorem 3 shows that the selected paths $DP(q)$ are disjoint, and if $maxL = 2$, then $DP(q)$ is compact.

THEOREM 3. *Given P_q, the set of q simple paths of size up to maxL-edges. The set $DP(q)$ returned by Algorithm 4 is the set of disjoint paths covering q. If $maxL = 2$, then $DP(q)$ is compact.*

PROOF: A path of the largest length $p \in P_q$ is inserted into $DP(q)$ and removed from q' (lines 5-6) if it fully exists in q', i.e., if $p \subseteq q'$ (line 4). This guarantees that all selected paths do not share any edge, i.e., they are disjoint.

Suppose that $DP(q)$ is not compact and $maxL = 2$. Then, there exist at least two 1-edge paths p_1 and p_2 in $DP(q)$ such that the path $p = p_1 \cup p_2$ is not selected by the algorithm. Since $p_1 \subseteq q'$ and $p_2 \subseteq q'$, then the only reason to not select p is that p disconnects q'. On the other hand, since removing p_1 or p_2 leaves q' connected, then removing p also leaves q' connected, i.e., p should have been selected, a contradiction. ■

Algorithm 4: `Cover`(q, P_q)

```
Input: $P_q$: $q$'s simple paths of size up to $maxL$-edges;
Output: $DP(q)$: disjoint paths covering $q$, initialized empty;

1:  Sort $P_q$ in decreasing order based on path size;
2:  $q' = q$;
3:  for each $p \in P_q$ do
4:      if $p \subseteq q'$ and $q' \setminus p$ is connected then
5:          Remove $p$ from $q'$;
6:          $DP(q) = DP(q) \cup \{p\}$;
7:      if $q'$ is empty graph then break;
8:  return $DP(q)$;
```

Algorithm 5: `Order`$(V_q, DP(q))$

```
Input: $DP(q) = \{p_1, p_2, \dots, p_{|DP(q)|}\}$;
Output: An order of $DP(q) = \{p'_1, p'_2, \dots, p'_{|DP(q)|}\}$;

1:  for each $u \in V_q$ do calculate $freq(u)$;
2:  $p'_1 = p_k$, $k = \text{argmax}_{p \in DP(q)} \sum_{u \in V_p} freq(u)$;
3:  $DP(q) = DP(q) \setminus \{p'_1\}$;
4:  $newDP(q) = \{p'_1\}$;
5:  $V = V_{p'_1}$;
6:  for $i = 2 \dots (|DP(q)| - 1)$ do
7:      $p'_i = p_k$, $k = \text{argmax}_{p \in DP(q)} |V_p \cap V|$;
8:      $DP(q) = DP(q) \setminus \{p'_i\}$;
9:      $newDP(q) = newDP(q) \cup \{p'_i\}$;
10:     $V = V \cup V_{p'_i}$;
11: return $newDP(q)$;
```

Although $C(p_i)$ is minimized for each p_i based on the occurrences pruning method, the search order considered in Algorithm 3 is random, and can seriously slow down the algorithm. Query disjoint paths $DP(q)$ should be explored in the order that facilitates getting the utmost benefit of applying the condition at line 5. In other words, we have to follow an order which excludes false local matches p' as early as possible, saving much of the time that may be taken on false long partial mappings. A local match p' is false if it satisfies the conditions at line 13. When we maximize the node overlapping of a currently processing query path p_i with the previously explored ones, we, in fact, maximize the benefits of using the condition at line 13, and thus increase the likelihood that false local matches are detected early. Therefore, we adopt an ordering of $DP(q) = \{p_1, p_2, \dots, p_{|DP(q)|}\}$, such that the node overlapping of V_{p_i} is maximized with $\cup_{j<i} V_{p_j}$. And, the first path p_1 is chosen such that $\sum_{u \in V_{p_1}} freq(u)$ is maximum, where $freq(u)$ is the frequency of the node u with respect to $DP(q)$. Algorithm 5 outlines the idea.

Algorithm 6: `PathIndex Based Query Processing`

```
Input: $q$: query graph; PathIndex;
Output: the answer set $\mathcal{D}_q$, initialized empty.

1:  $\mathcal{F}_q = \{p \subseteq q : p \text{ is a simple path} \wedge |p| \leq maxL\}$;
2:  $\mathcal{F}'_q = \{< p, \mathcal{O}(p, q) >: p \in \mathcal{F}_q\}$;
3:  $DP(q) = \texttt{Cover}(q, \mathcal{F}_q)$;
4:  $DP(q) = \texttt{Order}(V_q, DP(q))$;
5:  for each $g_i \in \mathcal{D}$ do
6:      Load $\mathrm{M}^{\mathcal{F}}_{g_i}$;
7:      for each $p \in \mathcal{F}'_q$ do
8:          if $(|\mathcal{O}(p, q)| > |\mathcal{O}(p, g_i)|)$ || $p \notin \mathcal{F}_{g_i}$ then
9:              continue to $g_{i+1}$;
10:     Load $\mathcal{F}'_{g_i}$;
11:     $\mathcal{C} = \{\}$; $\mathcal{C} = \texttt{Filter}(\mathcal{F}'_{g_i}, \mathcal{F}'_q, DP(q), \mathcal{C})$;
12:     if $\mathcal{C} = \{\}$ then continue;
13:     $M = \{0, \dots, 0\}$;
14:     $H = \{0, \dots, 0\}$;
15:     if $\texttt{Verify}(DP(q), \mathcal{C}, M, H)$ then $\mathcal{D}_q = \mathcal{D}_q \cup \{g_i\}$;
16:Procedure $\texttt{Filter}(\mathcal{F}'_g, \mathcal{F}'_q, DP(q), \mathcal{C})$
17: for each $p \in \mathcal{F}'_q \wedge p \in \mathcal{F}'_g$ do
18:     Process $\mathcal{O}(p, q)$ and $\mathcal{O}(p, g)$ to construct $B_p(q, g)$;
19:     if $B_p(q, g)$ has no semi-perfect matching then
20:         $\mathcal{C} = \{\}$;
21:         break;
22:     if $p \in DP(q)$ then $\mathcal{C} = \mathcal{C} \cup \{C(p)\}$;
23: return $\mathcal{C}$;
```

5.2 Query Processing

The overall PathIndex based query processing is outlined in Algorithm 6. Given a query q, q's simple paths of size up to $maxL$-edges are enumerated as in PathIndex, and $\mathcal{F}'_q$ is constructed. A compact path cover to the query is obtained by calling the procedure *Cover* (Algorithm 4). To produce an effective ordering of the covering paths, the *Order* procedure (Algorithm 5) is then used. Algorithm 6 is an iterative algorithm. At iteration i (lines 6-15), the algorithm decides whether q is a subgraph of the graph g_i or not. To do so, g_i passes through two filters. The first filter, named *Filter 1*, is outlined by lines 7-9. In this filter, the graph g_i is pruned if there is a query path that is not in g_i or the number of path occurrences in q is greater than that of g_i. Since both $\mathrm{M}^{\mathcal{F}}_{g_i}$ and $\mathrm{M}^{\mathcal{F}}_q$ are required, $\mathrm{M}^{\mathcal{F}}_{g_i}$ is loaded into memory (line 6). The second filter, named *Filter 2*, is outlined by lines 11-12. Filter 2 applies on graphs g_i that survived the first filter. It is based on both $\mathcal{F}'_{g_i}$ and $\mathcal{F}'_q$; thus, $\mathcal{F}'_{g_i}$ must be first loaded into memory (line 10). The *Filter* procedure (lines 16-23) is then used to prune g_i or, otherwise, finds a set of compatible paths in g_i to each path in the query cover. These compatible paths are used by the *Verify* procedure (Algorithm 3) which outputs whether q is matched in g_i or not.

The salient aspects of PathIndex based query processing are 1) $\mathcal{F}'_{g_i}$ of graphs that survived Filter 1 represent the main bulk of accessed information from the Index, 2) graphs are not required in the verification; the mapping informations (occurrences details) are sufficient to answer the query, and 3) the filtering and verification processes are flexible; they can independently be implemented to work with 1-edge and/or 2-edges and/or ... and/or $maxL$-edges path features. Moreover, Filter 2 can directly work on query disjoint paths instead of all features of the query. This flexibility makes PathIndex able to adjust even when there is an explosion in the number of paths as in dense graphs.

6. EXPERIMENTAL EVALUATION

In this section we evaluate the performance of PathIndex on real and synthetic graphs. PathIndex is implemented in standard C++ with STL library support and compiled with GNU GCC. Experiments were run on a PC with Intel 3GHz dual Core CPU and 4G memory running Linux. In experiments we consider vertex/edge labeled graphs.

Datasets: The first real dataset, referred to as AIDS_40K, is the AIDS Antiviral screen database (http://dtp.nci.gov/). It consists of 40,000 graphs, and these graphs have 45 vertices and 47 edges on average. There are totally 61 distinct vertex labels in the dataset but the majority of these labels are H, C, O and N. The total number of distinct edge labels is 3. AIDS_10K is a subset that are randomly drawn from AIDS_40K. The synthetic datasets are generated using the synthetic graph data generator GraphGen [5]. The generator allows us to specify various parameters such as the average graph density D, graph size E and the number of distinct vertex/edge labels L. For example, Syn10K.E30.D5.L50 means that it contains 10,000 graphs; the average size of each graph is 30; the density of each graph is 0.5; and the number of distinct vertex/edge labels is 50. Five synthetic datasets with varying parameter values are used in experiments in order to see performance changes with varying parameter values.

In order to study the scalability of PathIndex against different dataset size, we use a large real chemical compound dataset, referred to as Chem_1M. Chem_1M is a subset of the PubChem database (ftp://ftp.ncbi.nlm.nih.gov/pubchem/), and consists of one million graphs. Chem_1M has 23.98 vertices and 25.76 edges on average. The number of distinct vertex and distinct edge labels are 81 and 3, resp. For this study, we derive subsets from Chem_1M, each one consists of N graphs and called Chem_N dataset. Note that the real datasets AIDS_10K and Chem_1M, and the synthetic datasets are the same as that used in [13]. AIDS_40K is the same as that used in [10].

Query sets: There are six query sets Q4, Q8, Q12, Q16, Q20 and Q24. Each Qi consists of 1000 queries, each of which of size i. For AIDS_40K, we adopt the query set from [3]. To generate query sets for other datasets, a set of 1000 graphs whose size is larger than or equal to 24 are randomly selected from the dataset. Then, edges are removed from graphs such that the remaining graphs still remain connected. These graphs constitute Qi when all graphs are of size i.

Table 1: # Paths and $|\text{NH}_{\mathcal{D}}|$ in different datasets

Dataset	1-Edge	2-Edges	3-Edges	$\lvert\text{NH}_{\mathcal{D}}\rvert$
AIDS_40K	330	1476	4005	1350
Chem_1M	628	1369	2874	1475
Syn10K.E30.D3.L50	2296	12409	49767	15998
Syn10K.E30.D5.L50	13606	119148	-	46200
Syn10K.E30.D7.L50	17207	277357	-	69694

6.1 Performance Study

We evaluate the offline and online performance of PathIndex, and compare it against the state-of-the-art feature-based indexes such as FG-Index [5, 14], SwiftIndex [8] and CT-Index [10]. FG-Index and SwiftIndex are based on feature selection, whereas CT-Index exhaustively enumerates trees and cycles of limited size. FG-Index indexes frequent subgraphs based on the idea of δ-Tolerance closed frequent subgraphs, and uses a verification-free strategy to answer queries that are frequent in the dataset[2]. But, SwiftIndex uses frequent trees instead. As GraphGrepSX [15] (an approach using exhaustive enumeration of paths) and GCoding [16] (an indexing approach using neighborhood of database nodes) are outperformed by CT-Index, in this paper, we excluded both of them from comparisons. The executables for competitor methods were obtained from their authors. Hereafter, PathIndex, FG-index, SwiftIndex and CT-Index are abbreviated as PI, FG, SI and CT, resp.

In experiments, we adopt the default parameter values used in each technique: for FG, δ and σ are set to 0.1. For PI, we have experimented with different $maxL$, where $maxL \leq 3$. Table 1 shows the number of 1-edge, 2-edges and 3-edges paths in different datasets. It is noticed that when the graph density increases, the number of distinct paths increases as well. Recall that the real datasets are composed of sparse graphs and, thus, they are characterized by small number of paths. Since indexing very large number of paths is expensive, 1-edge paths as features are indexed by PathIndex for synthetic datasets and paths of size up to 3-edges are indexed for real datasets. Generally, PI adapts to 1-edge and/or 2-edges and/or 3-edges path features when the number of those features is acceptable. Table 1 also shows $|\text{NH}_{\mathcal{D}}|$, the number of distinct neighborhoods in every dataset. The same trend occurs: The number of distinct neighborhoods increases as graph density increases.

6.1.1 Verification Performance

Figure 5 demonstrates the benefit of using a path-at-a-time verifier (Algorithm 3). Our verifier is tested against the state-of-the-art graph matching algorithms like QuickSI [8] and Vflib (http://amalfi.dis.unina.it/graph/db/vflib-2.0) on AIDS_10K and Syn10K.E30.D5.L50 by varying the query size. QuickSI is a vertex-at-a-time verifier, whereas Vflib is a state space search algorithm. QuickSI significantly improves the Ullman algorithm by using the label and edge frequencies in the dataset to determine an effective search order of the search space.

In this experiment, two implementations of our verifier are used. The first is based on 1-edge paths of the query and called PV_1. The second, called PV_2, uses paths of size up

[2] Notice that the competitor FG-Index is the latest release that uses an additional indexing component called FAQ-index. The FAQ-index is dynamically constructed from the set of frequently asked non-frequent subgraphs. Hence, verification is also not required for processing frequently asked non-frequent queries.

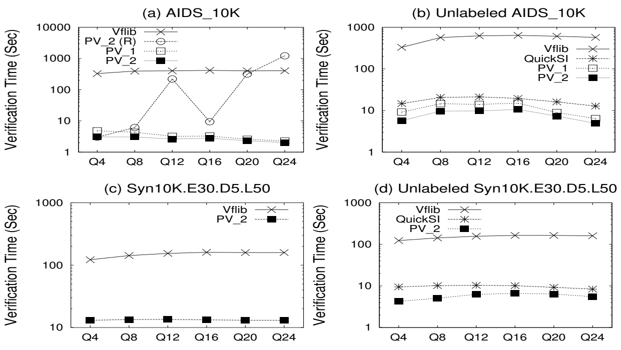


Figure 5: Verification Performance

to 2-edges. We also use PV_2 and PV_2(R) to denote our verifier with query disjoint paths ordered according to Algorithm 5 and randomly respectively. Note that, this experiment is performed on all data graphs, i.e., no filter is used. And, the reported time for our verifier also includes the time used for building query disjoint paths and their compatible graph paths from scratch. Figures 5(a),(c) report the results on vertex/edge-labeled graphs, whereas Figures 5(b),(d) report on unlabeled-edge graphs. First, note that the path order determined by Algorithm 5 makes our verifier (PV_2) faster than that with random order (PV_2(R)), and less sensitive to query size. Figures 5(a)-(d) show that both PV_1 and PV_2 perform the best. PV_2 significantly outperforms Vflib by one to two orders of magnitudes (Figure 5(a),(c)), and by 15 times to two orders of magnitudes (Figure 5(b),(d)). It also outperforms QuickSI by 2 to 4 times. These results are realistic since our verifier uses large-size local matches, an effective occurrences pruning method, and an effective search order to confine the search space. In the following experiments, PV_2 and PV_1 are adopted for real datasets and synthetic datasets, respectively.

Table 2: Offline Costs.

Dataset	Time (Sec)			Index Size (MB)			# Features	
	FG	CT	PI	FG	CT	PI	FG	PI
AIDS_40K	1347	451	9.49	158.2	49.1	25.8	5345	5781
Chem_100K	111	180	6.5	82.2	82	27.4	283	3434
Chem_200K	175	fail	13.6	171.7	fail	65.8	299	3851
Chem_300K	fail	fail	21.8	fail	fail	100.8	fail	4349
Chem_1M	fail	fail	82.5	fail	fail	382.1	fail	4871
SynD5.L50	21	7939	0.4	9.8	9	2.8	31	13606

fail: Algorithm fails to run on our machine.

6.1.2 Cost of Offline Processing

The offline performance in terms of index construction time and index size for AIDS_40K, Chem_100K, Chem_200K, Chem_300K, Chem_1M and Syn10K.E30.D5.L50 (SynD5.L50) are listed in Table 2.

Table 2 shows that PI construction time is much better than those of other indexing methods for all datasets. Moreover, it shows that PI construction algorithm is less sensitive to dataset size (the trend is approximately linear), and outperforms FG and CT algorithms by over an order of magnitude. This is because no data mining and expensive isomorphism tests are required for PI construction; only paths of small size are generated. Though CT does not perform the expensive mining process, its construction time on synthetic data is much slower than PI and FG. This is due to the large number of trees and cycles that have been enumerated and tested for isomorphism.

Contrasting indexes based on size, we note that PI has the smallest size. Though PI has more features than FG, its size is smaller than half of FG size; thanks to the small features (paths that compresses many of its supergraph features), the $NH_{\mathcal{D}}$ structure and the Accessibility Bridge arrays. Thanks also to our verification method which does not require the graphs while verification. Therefore, PI size does not include dataset size. Finally, CT has comparable size with FG for all datasets except for AIDS_40K, CT size is smaller than FG size.

Note that, the construction time and index size for FG and CT are not shown in Table 2 for larger Chem_N datasets (N > 200K for FG and N > 100K for CT) because their construction algorithms fail to run on our machine for large datasets.

Table 3: The pruning power of Filter 1 and Filter 2 of PathIndex on AIDS_10K dataset and their times

Filter	Q4		Q8		Q12		Q16		Q20		Q24	
	cands.	time	cands.	time	cands.	time	cands.	time	cands.	time	cands.	time
${F_1}^{1+2+3}$	2607	0.19	543.8	0.17	124.4	0.19	41	0.1	20.1	0.14	9.5	0.09
${F_2}^{1+2+3}$	2305	4.78	254.1	2.11	37.9	0.79	14.9	0.41	7.9	0.3	4.8	0.25
${F_2}^{3}$	2315	1.32	257	0.81	39.8	0.3	16.2	0.13	8.2	0.14	4.8	0.1
${F_2}^{2d}$	2412.7	1.07	362.5	0.42	67.3	0.11	26.1	0.05	13.1	0.07	6.7	0.03
Ans. Set	2303.6	——	210.8	——	26.4	——	10.1	——	5.7	——	3.9	——

cands.: Average candidate size per query.

time: The total running time (in Sec.) of 1000 queries in each query set.

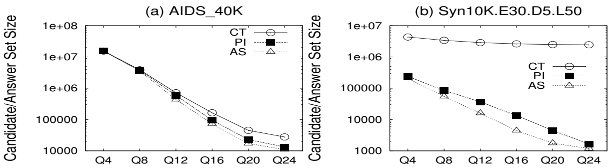


Figure 6: Candidate Size

6.1.3 Filtering Performance

In order to show the influence of Filter 1 and Filter 2 used with PI, Table 3 shows the number of candidates pruned by each filter and its running time for different query sets on AIDS_10K. In this experiment, $maxL = 3$. Also, Filter 1, abbrev. ${F_1}^{1+2+3}$, and Filter 2, abbrev. ${F_2}^{1+2+3}$, are based on the information of all features, i.e., 1-edge, 2-edges and 3-edges paths. Table 3 shows that the pruning power of Filter 1 is very high. It prunes most of the graphs in no time. On Q4, which contains small queries, for example, it prunes 75% of the graphs in 0.19 sec. Filter 1 achieves that speed because it performs at most two integer comparisons for each feature contained by the query and each graph. Although Filter 2 is applied on fewer graphs, it is rather slower since more computations are relatively required to first construct the bipartite matrix and then perform a semi-perfect matching test. Nevertheless, Filter 2 pruning is effective; Table 3 shows that it prunes almost all false graphs, i.e., most of the survived graphs are actual answers. Since much of the computations are repeated in Filter 2 due to the inclusion relationship between paths, we ran Filter 2 on 3-edges paths only, abbrev. ${F_2}^3$. Note that, every 3-edges path represents a maximal feature and contains five other features. Table 3 shows that ${F_2}^3$ is faster than ${F_2}^{1+2+3}$ and, on the other hand, achieves approximately the same pruning power on all query sets. For example, ${F_2}^3$ spends 1.32 sec compared to

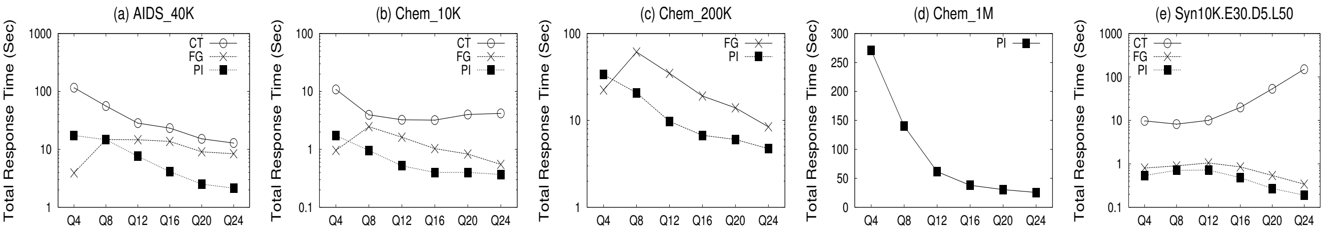


Figure 7: Total Response Time for Real and Synthetic Datasets

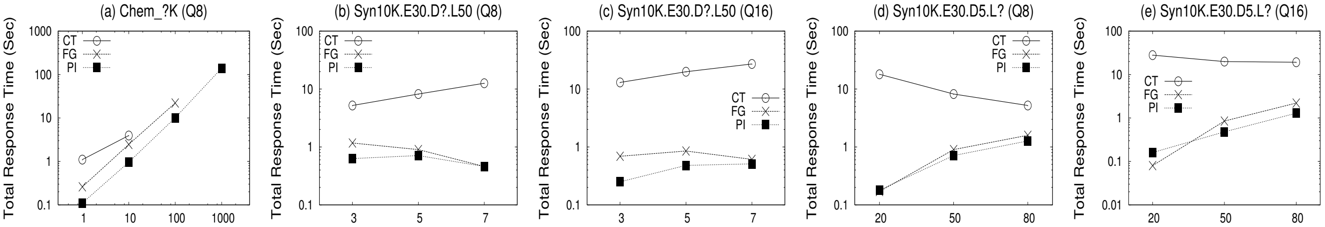


Figure 8: Scalability on Dataset Size(# Graphs in K) [a], Graph Density [b, c], and # Labels [d, e].

4.78 sec by ${F_2}^{1+2+3}$ on Q4 to prune almost the same number of graphs. Table 3 also shows that running Filter 2 on the query disjoint 2-edges paths (abbrev. ${F_2}^{2d}$) is faster than ${F_2}^3$ and relatively has acceptable pruning power.

Figure 6 plots the answer set size (AS), and the number of candidates returned by PI and CT on the real and synthetic datasets for the different query sets. The candidate size for FG is not shown because FG's executable does not support this feature; in FG, the filtering and verification processes are implemented together. Figure 6 shows that the pruning power of CT is comparable with that of PI on real datasets (sparse data) for moderate size queries Q4-Q12. For larger query size Q16-Q24, PI becomes better. On the synthetic dataset (dense data), the pruning power of CT deteriorates and the gap between CT and PI becomes very large. Thus, we conclude that PI has good pruning power on sparse as well as dense data.

6.1.4 Cost of Online Processing

Figure 7 reports the total response time comparing PI, FG and CT algorithms on different real and synthetic datasets. Note that, in experiments, to facilitate direct application of the PV_2 verifier on real datasets and the PV_1 verifier on synthetic datasets, the PI algorithm utilizes the 2-edges and 1-edges disjoint paths of each query for filtering respectively.

Figure 7 shows that PI has the best performance on all datasets. On AIDS_40K, PI outperforms CT by up to an order of magnitude for all query sets, and FG by up to a factor four, except on Q4, where FG does the best. On Chem_10K, the performance of PI relative to CT remains as for AIDS_40K; but relative to FG, we find that the gap increases as query size decreases. On the larger Chem_200K, the performance gap between FG and PI increases for all query sets, but FG still wins for Q4. On the largest Chem_1M dataset, PI is the only algorithm shown; other algorithms failed to run on our machine, and thus are not shown. FG is attractive for very small queries since a large amount of candidates can be verified without subgraph isomorphism testing, whereas for larger queries, the verification free technique can not take effect on most candidates. Figure 7 also shows that PI has the best performance on Syn10K.E30.D5.L50. It outperforms CT by up to three orders of magnitude. FG does not perform well for Q4 since there exist no frequent features of size 4 and thus the verification-free strategies are not used in this case.

6.1.5 Scalability Test

In order to test the scalability of PI against the dataset size, we ran PI and other methods on the generated subsets of Chem_1M. Figure 8(a) shows the total response time for Q8 on the generated subsets of Chem_1M. It shows that PI scales gracefully and outperforms FG and CT by over 2 and 3 times, respectively. FG and CT are not shown on larger subsets for the same previously mentioned reason. Figures 8(b)-(c) and 8(d)-(e) show the effects of changing graph density and the number of vertex/edge labels respectively for the query sets Q8 and Q16. For Q8, PI is comparable to FG on the two parameters. The performance gain increases as the number of labels increases and graph density decreases. For Q16, the same trend occurs with more performance gain on both parameters. CT is very sensitive to these parameters, displaying worse performance especially with large density and few labels.

Table 4: Offline Costs (unlabeled-edge graphs).

Dataset	Time (Sec)		Size (MB)		# Features	
	SI	PI	SI	PI	SI	PI
Unlabeled AIDS_40K	3965	7.23	207.3	22.7	3092	3930
Unlabeled SynD5.L50	0.92	0.3	4.25	2.4	60	620

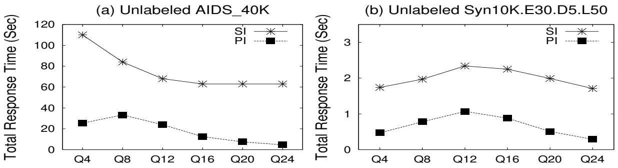


Figure 9: Response Time for Unlabeled-Edge Graphs

6.1.6 Performance on unlabeled-edge graphs

Here, we compared PI with SwiftIndex (SI) [8] in order to show the performance on unlabeled-edges graphs. Note that we carried out this experiment since SI is working on this type of graphs only, and uses the QuickSI verifier. We used AIDS_40K and SynD5.L50 after removing the edge labels, and we denoted them as Unlabeled AIDS_40K and Unlabeled SynD5.L50, respectively. Table 4 shows the superiority of PI on all offline costs. Figure 9 shows that PI outperforms SwiftIndex on both datasets. On sparse data (Unlabeled AIDS_40K), PI beats SI by up to an order of magnitude, whereas on dense data (Unlabeled SynD5.L50), the performance gain is lower.

7. RELATED WORK

Indexing neighborhood and utilizing it for pruning has been employed by many graph indexing methods [17, 16, 18, 19]. These methods can be classified as **node-based** indexing techniques since they index distinct database nodes with their local structures. In [17], the local structure is taken as the induced subgraph of the node and its neighbors, whereas, in [16], it is captured by the level-k path tree. [19] encodes all shortest paths information in the k-neighborhood subgraph of every database node in what is called *neighborhood signature*. In [18], a structure called Hybrid Neighborhood Unit (HNU) is devised. It stores for each node v in the graph its label, degree, v's neighbors and v's neighbors' neighbors. Different from these methods, our approach encodes neighborhood of features which are general subgraphs. Thus our approach can be considered as a hybrid between node-based and feature-based approaches, which scales both of them.

The main problem with node-based methods is the high cost of comparing large-size local structures. Node-based methods avoid this problem by indexing at most two-levels of neighborhood information of each database node. Our approach, on the other hand, indexes up to $(l+1)$-levels of neighborhood information of each node of an indexing feature, where l is the length of the longest path in that feature starting at the given node. Since PathIndex exhaustively enumerates paths, then each database node appears in at least one path feature. Thus, up to $(maxL+1)$-levels of neighborhood information of each database node are indexed in PathIndex.

Very few approaches try to reuse information from the filtering step to speed up the verification phase [8, 10, 2, 20]. GraphGrep [2] used the location information of each path in the candidate graph to prune parts of the graph which do not contain any path of the query. Though graphGrep leads to an improved verification phase, its filtering is limited. In contrast to GraphGrep, the SING approach [20] used path locality [3] to improve both verification and filtering. In [8], a verifier called QuickSI is introduced. QuickSI achieves good performance by using label and edge frequencies in the database to confine the search space. On the contrary, we develop a novel path-at-a-time verifier that benefits from the occurrences pruning method to reduce the search space, and employs a novel search order of query paths to speed up the matching process.

8. CONCLUSIONS

In this paper, to cope with the subgraph search problem, we proposed an efficient feature-based indexing approach. For a long time, it is believed that there is a tradeoff between the size of the feature set and its filtering strength. The presented approach broke this belief by compressing multiple features into one feature with some neighborhood information encoded. Neighborhoods are further used to guide the verification process, and a novel verification method is introduced. Extensive experiments demonstrate that our approach scale current graph indexing methods for subgraph search problem.

9. ACKNOWLEDGMENTS

The authors would like to thank James Cheng for providing FG-index, and Nils Kriege for providing CT-index , and Haichuan Shang for providing SwiftIndex and QuickSI. The authors also acknowledge the valuable comments of Xifeng Yan.

10. REFERENCES

[1] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Comleteness.* W. H. Freeman Co., 1979.

[2] R. Giugno and D. Shasha, "Graphgrep: A fast and universal method for quering graphs," *Proc. of the 16th International Conference on Pattern Recognition*, pp. 112–115, 2002.

[3] X. Yan, S. Yu, and J. Han, "Graph indexing: a frequent structure-based approach," *SIGMOD*, pp. 335–346, 2004.

[4] H. He and A. K. Singh, "Closure-tree: An index structure for graph queries," *ICDE*, pp. 38–49, 2006.

[5] J. Cheng, Y. Ke, W. Ng, and A. Lu, "Fg-index: towards verification-free query processing on graph databases," *SIGMOD*, pp. 857–872, 2007.

[6] S. Zhang, M. Hu, and J. Yang, "Treepi: A novel graph indexing method," *ICDE*, pp. 966–975, 2007.

[7] P. Zhao, J. X. Yu, and P. S. Yu, "Graph indexing: tree + delta <= graph," *VLDB*, pp. 938–949, 2007.

[8] H. Shang, Y. Zhang, and X. Lin, "Taming verification hardness: an efficient algorithm for testing subgraph isomorphism," *PVLDB*, pp. 364–375, 2008.

[9] D. W. Williams, J. Huan, and W. Wang, "Graph database indexing using structured graph decomposition," *ICDE*, pp. 976–985, 2007.

[10] K. Klein, N. Kriege, and P. Mutzel, "CT-index: Fingerprint-based graph indexing combining cycles and trees," *ICDE*, pp. 1115–1126, 2011.

[11] X. Yan, F. Zhu, P. S. Yu, and J. Han, "Substructure similarity search in graph databases," *SIGMOD*, pp. 766–777, 2005.

[12] J. R. Ullmann, "An algorithm for subgraph isomorphism," *ACM*, vol. 23(1), pp. 31–42, 1976.

[13] W.-S. Han, J. Lee, M.-D. Pham, and J. X. Yu, "igraph: a framework for comparisons of disk-based graph indexing techniques," *PVLDB*, pp. 449–459, 2010.

[14] J. Cheng, Y. Ke, W. Ng, and A. Lu, "Efficient query processing on graph databases," *TODS*, vol. 34, 2010.

[15] V. Bonnici, A. Ferro, R. Giugno, A. Pulvirenti, and D. Shasha, "Enhancing graph database indexing by suffix tree structure," *Recognition in Bioinformatics*, pp. 1195–203, 2010.

[16] L. Zou, L. Chen, J. X. Yu, and Y. Lu, "A novel spectral coding in a large graph database," *EDBT*, pp. 181–192, 2008.

[17] Y. Tian and J. Patel, "Tale: A tool for approximate large graph matching," *ICDE*, pp. 963–972, 2008.

[18] S. Zhang, J. Yang, and W. Jin, "Sapper: Subgraph indexing and approximate matching in large graphs," *PVLDB*, 2010.

[19] P. Zhao and J. Han, "On graph query optimization in large networks," *SIGMOD*, 2010.

[20] R. Natale, A. Ferro, R. Giugno, M. Mongiovi, A. Pulvirenti, and D. Shasha, "Sing: Subgraph search in non-homogeneous graphs," *BMC Bioinformatics*, vol. 11, p. 96, 2010.

[3] the starting positions of all its occurrences