Efficient Algorithms for Generalized Subgraph Query Processing

Wenqing Lin, Xiaokui Xiao, James Cheng, Sourav S. Bhowmick

School of Computer Engineering, Nanyang Technological University, Singapore

{wlin1, xkxiao, assourav}@ntu.edu.sg j.cheng@acm.org

ABSTRACT

We study a new type of graph queries, which injectively maps its edges to paths of the graphs in a given database, where the length of each path is constrained by a given threshold specified by the weight of the corresponding matching edge. We give important applications of the new graph query and identify new challenges of processing such a query. Then, we devise the cost model of the branch-and-bound algorithm framework for processing the graph query, and propose an efficient algorithm to minimize the cost overhead. We also develop three indexing techniques to efficiently answer the queries online. Finally, we verify the efficiency of our proposed indexes with extensive experiments on large real and synthetic datasets.

Categories and Subject Descriptors

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

General Terms

Algorithms, Experimentation, Performance

Keywords

Graph Databases, Graph Matching Algorithm, Graph Indexing, Graph Querying

1. INTRODUCTION

Graph is a powerful data model that can naturally represent various entities and their relationships. Graph data is ubiquitous today and being able to query such graph data is beneficial to many applications. For example, in bio-informatics and chemical informatics, graphs can model compounds and proteins, and graph queries can be used for screening, drug design, motif discovery in protein structures, and protein interaction analysis. In computer vision, graphs represent organization of entities in images and graph queries can be used to identify objects and scenes. In heterogeneous web-based data sources and e-commerce sites, graphs model schemas and graph matching can be applied to solve problems of schema matching and integration. There are also many other applications, such as program flows, software and data engineering, taxonomies, etc., where data is modeled as graphs and it is essential to search and query the graph data.

Existing research focuses on mainly two types of graph datasets, one consisting of a single large graph (e.g., an online social network or the entire citation graph in a certain domain) and the other consisting of a large set of small or medium-sized graphs. We focus on the later, which are also very popular in real life (e.g., most of the examples we listed earlier belong to this type).

To query a graph database, $\mathcal{G}$, that consists of many small graphs, there are three types of queries commonly studied in the literature. Let q be a query graph. The first one is *subgraph query* [4, 9, 13, 15, 18, 21, 23], which finds the subset of graphs $\mathcal{A}$ of $\mathcal{G}$ such that q is a *subgraph* of any graph in $\mathcal{A}$. The second one is *supergraph query* [2, 3, 14, 22], which finds the subset of graphs $\mathcal{A}$ of $\mathcal{G}$ such that q is a *supergraph* of any graph in $\mathcal{A}$. The third one is *similarity query* [12, 19, 20, 24], which finds the subset of graphs $\mathcal{A}$ of $\mathcal{G}$ such that q is a *similar graph* of any graph in $\mathcal{A}$ according to a given similarity measure.

The three types of queries are useful in different applications. However, both subgraph queries and supergraph queries are too rigid and therefore similarity queries are proposed as an alternative. Existing similarity queries are mostly measured by the edit distance [20, 24] or maximum common subgraph [12, 19], which is reasonable for some applications but often fails to capture meaningful patterns or targets in applications where critical objects or entities may have to be matched or they may be within a distance from each other that is beyond the specified similarity distance (i.e., the similarity threshold). We show such an application, where both exact queries and similarity queries are not applicable, by the following example.

Example 1. *Consider a drug design system, which supports the inventive process of finding new medications based on the knowledge of the biological target. Figure 1 shows some compounds in the database, i.e., g_1, g_2, and g_3. A compound can be naturally modeled as a graph, where atoms are vertices, the chemical name of the atom is the label of the correspond vertex, and the chemical bonds between any two atoms are modeled as edges in the graph. Among many drug design methods, the pharmacophore model is the most popular one whose goal is to find the substructures that are closely matched to the objective. ALADDIN [17] is a com-*

CIKM'12, October 29–November 2, 2012, Maui, HI, USA.

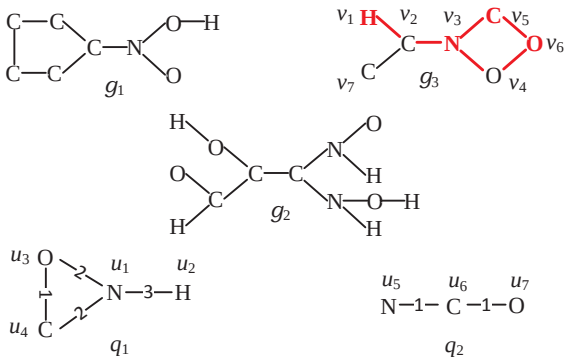


Figure 1: A drug database, $\mathcal{G} = \{g_1, g_2, g_3\}$, and two query graphs, q_1 and q_2

Table 1: A graph query in ALADDIN language

POINT N ; POINT H ; POINT O ; POINT C ;
DISTANCE (1, 2) 1 3 ; DISTANCE (1, 3) 1 2 ;
DISTANCE (1, 4) 1 2 ; DISTANCE (3, 4) 1 1 ;

puter program for the design and recognition of compounds that meet geometric, steric, and sub-structural criteria. ALADDIN also uses a precise geometric description language to define the properties of a designed molecule.

The query shown in Table 1 is written in the ALADDIN language, which is to find a graph pattern where

1. *there are four atoms:* `N`, `H`, `O`, *and* `C`, *whose positions are at* 1, 2, 3, *and* 4, *respectively;*
2. *the distance between* `N` *and* `H` *is 1 to 3, and similarly 1 to 2 between* `N` *and* `O`, *1 to 2 between* `N` *and* `C`, *and exactly 1 between* `O` *and* `C`.

Since the distance between all pairs of atoms can be estimated [11], the distance can be further modeled as the number of bonds that connect the atoms. Thus, the query in Table 1 can be converted to a graph query which finds all graphs in the database such that

1. *there exist four vertices u_1 to u_4 in the graph whose labels are* `N`, `H`, `O`, *and* `C`, *respectively;*
2. *let $P = \langle u_i, \ldots, u_j \rangle$ be a path that connects u_i and u_j and $|P|$ be the length of the path:*

   there exist paths $P_1 = \langle u_1, \ldots, u_2 \rangle$, $P_2 = \langle u_1, \ldots, u_3 \rangle$, $P_3 = \langle u_1, \ldots, u_4 \rangle$, and $P_4 = \langle u_3, \ldots, u_4 \rangle$, respectively, where $1 \leq |P_1| \leq 3$, $1 \leq |P_2| \leq 2$, $1 \leq |P_3| \leq 2$, and $|P_4| = 1$.

Such a query can be naturally represented as the query graph[1] *q_1 shown in Figure 1, and the answer to this query is $\{g_3\}$. In such a query, subgraph query cannot be applied, while similarity query is also not suitable when the matching paths are long.*

In this paper, we study this new type of graph queries as described in Example 1, which will be formally defined in Section 2. But intuitively, the new query is a generalization of the subgraph query, which generalizes exact edge matching to path matching constrained by a path length; that is, instead of matching each edge as in a subgraph query, we find a path with two matching end vertices for each edge in the query graph, where the length of the matching path must be within the specified edge weight. Thus, the new query has a much stronger expressive power than a subgraph query.

Such a query is also useful in many other applications. For example, in querying user online traversal graphs, one may be only interested in whether users have visited certain important sites within a certain number of clicks, while an exact or a quality similar matching may not exist. In searching pictures in an image database, it is often rare to find an exact or even similar matching due to the huge amount of irrelevant information in the background (note that similarity measure by edit distance or maximum common subgraph often counts all such irrelevant information in the matching); in this case, we can specify a few features to be focused in the matching while relaxing the links between the features by some reasonable edge weight.

Processing the new query, however, is significantly more challenging. For both subgraph and supergraph query processing, it involves subgraph isomorphism which is NP-hard. The relaxation in the new query from exact edge matching to approximate path matching essentially further explodes the already exponential search space. Existing pruning techniques cannot be directly applied or they are simply not adequate, since our generalized query graph is different from the indexed features. Therefore, this paper proposes new effective pruning techniques and efficient data structures to solve this challenging problem.

Our contributions. The contributions of this paper are four-fold. First, we propose the problem of generalized subgraph query processing, which is useful in applications where subgraph queries are too restrictive to apply while similarity queries may return low quality answers due to large edit distance arisen from abundant irrelevant information. Second, we devise a fast algorithm for generalized subgraph matching, which is a significantly more complicated matching problem than subgraph isomorphism. Third, we develop three indexes for the efficient processing of generalized subgraph queries, namely, a *distance-based* index, a *frequent-pattern-based* index, and a *star-structure-based* index. We discuss in details the strengths and limitations of the indexes. Fourth, we verify the efficiency of our matching algorithm (for candidate verification) and our indexes (for filtering) using both real and synthetic datasets.

Paper Organization. Section 2 gives the notations and formally defines the problem. Section 3 presents the generalized subgraph matching algorithm. Section 4 discusses in details the three indexes. Section 5 reports the experimental results. Section 6 discusses the related work and Section 7 concludes the paper.

2. PROBLEM STATEMENT

Let $\mathcal{G}$ be a database that contains a set of simple and labeled graphs. We denote each graph $g \in \mathcal{G}$ as a triplet $g = (V_g, E_g, l_g)$, where V_g and E_g are the sets of vertices and edges in g, respectively, and l_g is a labelling function that maps each vertex in g to a label in a finite alphabet. For ease of exposition, we assume that all edges in g are undirected; our results can be easily extended for directed graphs.

For any vertices u and v in a graph $g \in \mathcal{G}$, we define the *distance* between u and v, denoted as $dist_g(u, v)$, as the number edges in the shortest path between u and v. For instance,

[1] We assume that path length cannot be negative.

in the graph g_3 in Figure 1, we have $dist_{g_3}(v_1, v_3) = 2$, since the shortest path between v_1 and v_3 contains two edges (v_1, v_2) and (v_2, v_3).

We aim to support *generalized subgraph queries* on $\mathcal{G}$. In particular, a *generalized subgraph* q is a simple, undirected, and labelled graph where each edge carries a positive integer weight. We denote q as a quadruple (V_q, E_q, l_q, t), where V and E are the sets of vertices and edges in q, respectively, l_q is the labelling function for q, and t is a function that maps each edge in q to its weight. We say that a graph $g \in \mathcal{G}$ *matches* q, if there exists an injective function f from V_q to V_g, such that for any edge (u, v) in q, (i) the labels of u and $f(u)$ are the same, (ii) the labels of v and $f(v)$ are the same, and (iii) the distance between $f(u)$ and $f(v)$ in g is no more than the weight of (u, v).

For example, in Figure 1, the graph g_3 matches the generalized subgraph q_1. To explain this, let us consider an injective function f that maps u_1 to v_3, u_2 to v_1, u_3 to v_6, and u_4 to v_5. For the edge (u_1, u_2) in q_1, we have $f(u_1) = v_3$ and $f(u_2) = v_1$, and the distance $dist_{g_3}$ between v_3 and v_1 in g_3 equals 2, which is no more than the weight associated with (u_1, u_2). The cases for the other edges in q_1 can be verified in a similar manner.

Given a generalized subgraph q, a generalized subgraph query on $\mathcal{G}$ returns the graphs in $\mathcal{G}$ that match q. For convenience, we refer to q as the *query graph*, and the graphs in $\mathcal{G}$ as the *data graphs*. In addition, we say that a data graph g *contains* q (denoted by $q \subseteq g$), if g matches q.

3. GENERALIZED SUBGRAPH MATCHING ALGORITHM

To enable generalized subgraph queries on $\mathcal{G}$, we need to first address a crucial problem: *How do we decide whether a data graph $g \in \mathcal{G}$ matches the query graph q?* We refer to this problem as the *generalized subgraph matching* problem. It is not hard to see that this problem is NP-hard; in particular, when the weights of all edges in q equal 1, testing whether a data graph g matches q is equivalent to the *subgraph isomorphism* problem, which has been shown to be NP-complete [5].

Given that generalized subgraph matching is theoretically intractable, we resort to heuristics and propose a solution that provides practical efficiency. The core of our solution is a cost-based matching approach that significantly extends and improves the existing heuristic algorithms [13, 16] for the subgraph isomorphism problem. In what follows, we will first introduce the existing methods for subgraph isomorphism (in Section 3.1), and then present the details of our solution (in Section 3.2).

3.1 Existing Algorithms for Subgraph Isomorphism

The classic solution for the subgraph isomorphism is Ullmann's algorithm [16], which matches the vertices in the query graph q to the vertices in the data graph g in an iterative manner. Specifically, in each iteration, the algorithm selects an unmatched vertex u in q, maps it to an unmatched vertex in g with the same label, and then checks whether the mapping is *feasible*, i.e., whether any two matched vertices in q that induce an edge in q are mapped to two vertices in g that induce an edge in g. If the mapping is feasible, the algorithm will enter the next iteration to match the remaining vertices in q. Otherwise, the algorithm will try matching u to another unmatched vertex in g. If there is no vertex that u can be matched to, the algorithm backtracks to the last matched vertex u' in q, re-maps u' to an unmatched vertex in g, and then re-starts the current iteration.

For example, in Figure 1, given the query graph q_2 and the data graph g_3, Ullmann's algorithm may first map u_5 to v_3, and then map u_6 to v_5. In that case, u_5 and u_6 induce an edge in q_2, while v_3 and v_5 also induce an edge in g_3, i.e., the matching is feasible. Assume that, in the next iteration, the algorithm maps u_7 to v_4. Then, u_6 and u_7 induce an edge in q_2, but the vertices that they are mapped to (i.e., v_5 and v_4) do not induce any edge in g_3. As a consequence, the mapping is infeasible, and hence, the algorithm would proceed to re-map u_7 to another unmatched vertex in g_3.

Intuitively, the efficiency of Ullmann's algorithm depends on the order in which the vertices in q are matched. For instance, assume that q contains only two vertices u_1 and u_2, such that u_1 has the same label with only one vertex v_1 in the data graph g, whereas u_2 has the same label with almost all vertices in g. If we invoke Ullmann's algorithm and map u_1 to v_1 in the first iteration, then in the remaining iterations, we only need to examine whether u_2 can be mapped to a vertex adjacent to v_1. In contrast, if the first iteration maps u_2 (instead of u_1) to some vertex v in g, then in the remaining iterations, we not only need to try mapping u_1 to the neighbors of v, but also need to consider other possible mappings that match u_2 to other vertices in g, i.e., the search space of the algorithm becomes significantly larger.

Despite the importance of vertex mapping order, it is not taken into account in Ullmann's algorithm. This motivates a more advanced method called *QuickSI* [13], which improves over Ullmann's algorithm by heuristically choosing a mapping order that is likely to reduce computation cost. Specifically, QuickSI decides the vertex mapping order based on two sets of statistics pre-computed from the graph database $\mathcal{G}$. First, for any vertex u that can possibly appear in a query graph, QuickSI pre-computes its *frequency* in $\mathcal{G}$, i.e., the average number of vertices in each data graph (in $\mathcal{G}$) that have the same label with u. Second, for any edge e that may appear in a query graph, QuickSI also pre-computes its frequency in $\mathcal{G}$, i.e., the average number of edges in each data graph that have endpoints with labels matching those of the endpoints of e. With these statistics, for any given query graph q, QuickSI first generates a spanning tree of q, such that vertices and edges closer to the root of tree tend to have lower frequencies in $\mathcal{G}$. After that, QuickSI generates an ordering of the vertices in q following a traversal of the spanning tree that recursively visits the branch with the least frequent edge. The resulting vertex order is then used whenever QuickSI compares a data graph g with q. Intuitively, this vertex order improves efficiency, as it tends to ensure that the search space of the matching algorithm would be reduced significantly after each iteration.

3.2 A Cost-based Approach for Generalized Subgraph Isomorphism

Both Ullmann's algorithm and QuickSI can be extended for generalized subgraph isomorphism, with a modified feasibility check in each iteration. Specifically, each time after we map a vertex u in the query graph q to a vertex v in the data graph g, we would decide whether the mapping is feasible by examining every edge e in q that is induced by u and any vertex u' in q that has been matched. Let

v' be the vertex in g that u' is mapped to. If for each e, the distance $dist_g(v, v')$ between v and v' is no more than the weight $w(e)$ of e, then the mapping is feasible, and we would proceed to the next iteration. Otherwise, we would re-map u to other unmatched vertex in g; if there does not exist any feasible mapping for u, we would backtrack to the last matched vertex in q and re-map it (as with the case of subgraph isomorphism).

The aforementioned extensions of Ullmann's algorithm and QuickSI, however, leave much room for improvements. In particular, Ullmann's algorithm does not exploit the order of vertex mapping for efficiency; QuickSI heuristically tunes the vertex mapping order, but its tuning method is rather ah hoc and is without a formal model that justifies mapping a vertex ahead of any other. To remedy this, we propose a novel algorithm for generalized subgraph isomorphism that incorporates a cost model for selecting a preferable order of vertex mapping. In the following, we will first present the rationale behind our method, and then provide the details about our cost model and algorithm.

Assume that the query graph q and the data graph g contain m and n vertices, respectively. Totally, there exist $P(n, m) = n!/(n-m)!$ different ways to map the vertices in q to distinct vertices in g, and these $P(n, m)$ possible matchings constitute the search space for the generalized subgraph isomorphism algorithm. ($P(m, n)$ denotes the number of m-permutations of n.) To efficiently decide whether g matches q, it is essential that the algorithm should traverse the search space in a judicious order that enables it to pinpoint a solution (if any) as quickly as possible. This motivates us to match vertices in q in an order based on how likely they can reduce the search space that we need to explore. Note that we use the same node mapping order for all data graphs (as in QuickSI), so as to avoid the overhead of re-computing the node order for each data graph.

Specifically, to pick the first vertex in q to be matched, we would inspect each edge e in q, and examine the *frequency of* e (denoted as $c(e)$) in the data graphs in $\mathcal{G}$. The frequency of (u', u^*) in q is defined as the average number of vertex pairs (v', v^*) in each data graph in $\mathcal{G}$, such that (i) the labels of u' and v' are the same, (ii) the labels of u^* and v^* are the same, and (iii) the distance between v' and v^* is no more than the weight of (u', u^*). (To facilitate this step of the algorithm, we pre-compute the frequency of any edge that may appear in the query graph.)

For each e, we intuitively estimate that it can be matched to $c(e)$ vertex pairs in the data graph. Given this estimation, if a vertex u is an endpoint of e and we choose to match u first, then the search space size induced by mapping u can be estimated as $c(e) \cdot P(\bar{n} - 1, m - 1)$, where $\bar{n}$ denotes the average number of vertices in the data graphs. The rationale here is that u is expected to be mapped to around $c(e)$ vertices in a data graph, and the other unmatched $m-1$ vertices in q are expected to be matched to around $\bar{n} - 1$ vertices in g in $P(\bar{n} - 1, m - 1)$ different ways; therefore, the number of possible matchings that remain to explored can be estimated as $c(e) \cdot P(\bar{n} - 1, m - 1)$. Accordingly, we pick a vertex u incident to the edge e with the smallest $c(e)$, and set u as the first vertex to be matched. The term $P(\bar{n} - 1, m - 1)$ is ignored since its value is the same for all vertices in q. (This helps us avoid the pathological case when $\bar{n} < m$, in which case $P(\bar{n} - 1, m - 1)$ is undefined.) Given that the edge e with the smallest $c(e)$ has two endpoints, we choose the endpoint u with the smaller frequency $c(u)$.

The order of the remaining vertices is decided in a similar manner. Assume that we have picked a set S of k vertices and we are about to choose the next vertex to be matched. Let u' be any vertex that has not be selected. If u' is not connected to any vertex in S by an edge in q, then we estimate the search space size induced by mapping u' as

$$\min_{\text{any edge } e \text{ adjacent to } u'} c(e) \cdot N(S) \cdot P(\bar{n} - k - 1, m - k - 1), \quad (1)$$

where $N(S)$ denotes the number of ways to match the first k vertices, and $P(\bar{n} - k - 1, m - k - 1)$ is the number of ways to match the remaining $m - k - 1$ vertices except u'. As will be shown shortly, we do not need to compute the values of $N(S)$ and $P(\bar{n} - k - 1, m - k - 1)$.

On the other hand, if u' has some edges that are incident to the vertices in S, then our estimation of the search space size would take those edges into account. Let E be the set of edges in q that connect u' to the vertices in S. For each e in E that connects u' to a vertex u^*, we examine the frequency of u^* (denoted as $c(u^*)$) in $\mathcal{G}$, as well as the frequency of e (denoted as $c(e)$). Given $c(u^*)$ and $c(e)$, we intuitively estimate that the vertex u^* is connected to around $c(e)/c(u^*)$ vertices that have the same label with u'. Therefore, the search space size induced by mapping u' is estimated as

$$c(e)/c(u^*) \cdot N(S) \cdot P(\bar{n} - k - 1, m - k - 1), \quad (2)$$

where $\sum_{u \in S} c(u)$ and $P(\bar{n} - k - 1, m - k - 1)$ are as explained in Equation 1. We refer to $c(e)/c(u^*)$ as the *matching rate of* u' *implied by* e, and we denote it as $r(u', e)$.

Observe that each edge $e \in E$ may imply a different matching rate of u', leading to different estimations of the search space size. We combine all estimations by taking the smallest one, i.e., the size of the search space is estimated as

$$\min_{e \in E} r(u', e) \cdot N(S) \cdot P(\bar{n} - k - 1, m - k - 1). \quad (3)$$

For convenience, we let $r(u') = \min_{e \in E} r(u', e)$ if u' is connected to the vertices in S by at least one edge in the data graph, otherwise we let $r(u')$ be the minimum frequency of an edge in q that is adjacent to u'. Given Equations 1 and 4, we choose the next vertex u' to be matched as the one that minimizes the estimated search space size, i.e.,

$$u' = \arg\min_{u} \{r(u)\}. \quad (4)$$

Note that Equation 4 does not involve the terms $N(S)$ and $P(\bar{n} - k - 1, m - k - 1)$ (which appear in both Equations 1 and 4). This is because their values are the same for all possible u', and hence, they have no effect on the selection of u'.

In summary, our algorithm optimizes the vertex matching order by a qualitative prediction of how each vertex may help reduce the search space size. As will be shown in Section 5, our experimental results demonstrate the superiority of our algorithm over both Ullmann's algorithm and QuickSI on both standard and generalized subgraph isomorphism tests.

4. INDEXING TECHNIQUES

Although in Section 3 we proposed a reasonably fast algorithm for generalized subgraph matching, it is still impractical to answer a query by sequentially scanning the input database and matching the query graph with each

data graph, especially if the database is large. We apply the *filtering*-and-*verification* strategy to reduce the matching cost, that is, we first filter out as many unmatching data graphs as possible and then verify the remaining candidate data graphs by matching them with the query graph one by one. To do this, it is important to design an effective indexing technique to filter out the unmatching data graphs.

In this section, we propose three indexing techniques: **D-Index**, **FP-Index** and **S-Index**. First, in Section 4.1 we present D-Index, which can be easily constructed but its pruning power is relatively weak. Then, we propose FP-Index in Section 4.2, which has an expensive construction cost but is partially verification-free. Lastly, in Section 4.3 we propose S-Index, which explores the star structures to achieve effective pruning as well as a low construction cost.

4.1 Distance Index

We first present D-index, which is constructed based on the distance among pairs of vertices in each data graph. Given a data graph $g = (V_g, E_g, l_g) \in \mathcal{G}$, we obtain the *distance set* ($\mathcal{DS}$) of all triplets of every two vertices consisting of their ordered labels and the correspond distance in g as follows.

$$\mathcal{DS}(g) = \{(l_g(u), l_g(v), dist_g(u,v)) : u, v \in V_g, l_g(u) \leq l_g(v)\}$$

A distance triplet $(l_1, l_2, d) \in \mathcal{DS}(g)$ is subsumed by another distance triplet $(l_1, l_2, d') \in \mathcal{DS}(g)$ if $d > d'$. We say that a subset $\mathcal{DS}_{min}(g) \subseteq \mathcal{DS}(g)$ is minimal if each distance triplet in $\mathcal{DS}_{min}(g)$ is not subsumed by any other distance triplet, that is, for each $(l_1, l_2, d) \in \mathcal{DS}_{min}(g)$, there does not exist $(l_1, l_2, d') \in \mathcal{DS}_{min}(g)$ such that $d' < d$.

EXAMPLE 2. *Assume that vertex labels are ordered lexicographically, i.e.,* O<N<H<C. *Consider the data graph g_3 in Figure 1, the distance set of g_3 is $DS(g_3)$ = {(C,C,1), (C,C,2), (C,C,3), (H,C,1), (H,C,2), (H,C,3), (N,C,1), (N,C,2), (O,C,1), (O,C,2), (O,C,3), (O,C,4),(N,H,2), (O,H,3), (O,H,4), (O,N,1), (O,N,2), (O,O,1)}, and $\mathcal{DS}_{min}(g_3)$ = {(C,C,1), (H,C,1), (N,C,1), (O,C,1), (N,H,2), (O,H,3), (O,N,1), (O,O,1)}. Note that $|\mathcal{DS}(g_3)| = 18$ while $|DS_{min}(g_3)| = 8$.*

The minimal set of distinct distance triplets in the database is then given by

$$\mathcal{DS} = \cup_{g \in \mathcal{G}} \mathcal{DS}_{min}(g).$$

For each distance triplet $(l_1, l_2, d) \in \mathcal{DS}$, the set of data graphs that contain (l_1, l_2, d) is given by

$$\mathcal{A}(l_1, l_2, d) = \{g : (l_1, l_2, d) \in \mathcal{DS}_{min}(g)\}.$$

The *Distance Index* (*D-index*) is constructed on $\mathcal{DS}$ and $\mathcal{A}(l_1, l_2, d)$ for each $(l_1, l_2, d) \in \mathcal{DS}$, which is to be detailed as follows.

4.1.1 Index Construction

The structure of D-index consists of the following parts:

- A B+-tree index, called the *Label Pair Index* (LPI), stores all pairs of labels of the triplets in $\mathcal{DS}$.
- A sorted list of distance values for each label pair $(l_1, l_2) \in LPI$, denoted by $LPI(l_1, l_2).DV$, that is, $LPI(l_1, l_2).DV = \{d : (l_1, l_2, d) \in \mathcal{DS}\}$.
- Each distance value $d \in LPI(l_1, l_2).DV$ for any $(l_1, l_2) \in LPI$ is associated with a set of data graphs $\mathcal{A}(l_1, l_2, d)$, we further denote it as $LPI(l_1, l_2).DV(d) = \mathcal{A}(l_1, l_2, d)$.

Algorithm 1: BUILD-DINDEX($\mathcal{G}$)

input : the graph database, $\mathcal{G}$
output: the D-index, LPI

```
1 for g ∈ G do
2     Compute DS_min(g) ;
3     for (l1, l2, d) ∈ DS_min(g) do
4         LPI ← (l1, l2) ;
5         LPI(l1, l2).DV ← d ;
6         LPI(l1, l2).DV(d) ← g ;
7 return LPI
```

Algorithm 2: QUERY-DINDEX($q, LPI, \mathcal{G}$)

input : the query graph, $q = (V_q, E_q, l_q, t)$
D-Index, LPI
the graph database, $\mathcal{G}$
output: the candidate set of q, $\mathcal{C}(q)$

```
1 C(q) ← G ;
2 for (l1, l2, d) ∈ DS_min(q) do
3     C(l1, l2, d) ← ∅ ;
4     for k ∈ LPI(l1, l2).DV and k ≤ d do
5         C(l1, l2, d) ← C(l1, l2, d) ∪ LPI(l1, l2).DV(k) ;
6     C(q) ← C(q) ∩ C(l1, l2, d) ;
7 return C(q)
```

The algorithm for D-index construction, BUILD-DINDEX, is shown in Algorithm 1. For each data graph $g \in \mathcal{G}$, we first compute its minimal distance set $\mathcal{DS}_{min}(g)$ (Line 2). Then, for each distance triplet $(l_1, l_2, d) \in \mathcal{DS}_{min}(g)$, we assign (l_1, l_2) to LPI and put the distance value d to the sorted list $LPI(l_1, l_2).DV$ (Line 4-5). Finally, g is included in $LPI(l_1, l_2).DV(d)$ (Line 6).

4.1.2 Query Processing

Given a query graph $q = (V_q, E_q, l_q, t)$, we first obtain its minimal distance set $\mathcal{DS}_{min}(q)$ by all pairs shortest path algorithm. For each distance triplet $(l_1, l_2, d) \in \mathcal{DS}_{min}(q)$, the correspond candidate set $\mathcal{C}(l_1, l_2, d)$ can be obtained by merging all the graph sets associated with $LPI(l_1, l_2).DV(k)$ for $1 \leq k \leq d$. The final candidate set $\mathcal{C}(q)$ for verification is the intersection of all the candidate sets of each distance triplet $(l_1, l_2, d) \in \mathcal{DS}_{min}(q)$, that is

$$\mathcal{C}(q) = \cap_{(l_1, l_2, d) \in \mathcal{DS}_{min}(q)} \mathcal{C}(l_1, l_2, d).$$

As shown in Algorithm 2, processing the query graph q by D-index obtains the candidate set for each distance triplet $(l_1, l_2, d) \in \mathcal{DS}_{min}(q)$ (Lines 3-5), and then intersects them to output the candidate set of q (Line 6).

LEMMA 1. *Given a query graph $q = (V_q, E_q, l_q, t)$, its answer set $\mathcal{A}(q)$ is a subset of* BUILD-DINDEX($q, LPI, \mathcal{G}$).

PROOF. Consider a data graph $g = (V_g, E_g, l_g) \in \mathcal{G}$ that matches q. For each edge $(v, u) \in E_q$, we can map it to a path $P = \langle f(v), \ldots, f(u) \rangle$ such that $|P| \leq t(v, u)$. Without the loss of generality, we assume that $l_g(f(v)) \leq l_g(f(u))$. Consider the distance triplet $(l_g(f(v)), l_g(f(u)), d) \in \mathcal{DS}_{min}(g)$, we have $d \leq |P|$, that is $d \leq t(v, u)$, which completes the proof. □

4.1.3 Complexity Analysis

Assume that α is the average number of vertices and β is the average number of edges for the graphs in $\mathcal{G}$, the space complexity of D-index is $O(\alpha^2|\mathcal{G}|)$. Since the construction time for each $\mathcal{DS}_{min}(g)$ is $O(\alpha\beta)$ by starting a BFS from each vertex in g, the time complexity of constructing D-index is $O(\alpha\beta|\mathcal{G}|)$.

The generation of minimal distance set of query graph q can be done in $O(|V_q|^3)$ time. For each distance triplet in $\mathcal{DS}_{min}(q)$, the response time of D-index is $O(\log(md))$ where m is the number of distinct labels in $\mathcal{G}$ and d is the largest distance. Thus, the index response time for the graph pattern is $O(|V_q|^3 + k^2\log(md))$, where k is the number of distinct labels in q. Note that, in practice, both $|V_q|$ and k are very small.

4.2 Frequent Pattern Index

The shortcoming of D-index is that it loses the structural information of data graphs. As a result, the filtering is not effective enough, leading to a lot of unmatched candidate graphs. Although there are graph indexing approaches that retain structural information of data graphs [4, 9, 15, 18, 23], they cannot be directly applied to answer generalized subgraph queries.

In order to employ structural information for answering generalized subgraph queries, we propose the concept of *frequent generalized subgraph* (*FGG*) patterns and apply FGGs to design a structural index called *Frequent Pattern Index* (*FP-index*). The challenges, however, are 1) how to efficiently mine the FGGs, 2) how to apply and index FGGs for filtering. We address the two challenges as follows.

The first challenge can be addressed by the pattern-growth approach [1]. The difference is that in an FGG, edges are weighted. To obtain weighted edges for FGGs, we grow the frequent patterns from weighted edges. We initialize the set of weighted edges by taking the set of distinct distance triplets $\mathcal{E} = \cup_{g\in\mathcal{G}}\mathcal{DS}(g)$ introduced in Section 4.1, where each distance triplet $(l_1, l_2, d) \in \mathcal{E}$ is considered as an edge (l_1, l_2) with weight d, while $|\mathcal{C}(l_1, l_2, d)|$ is the frequency of the edge.

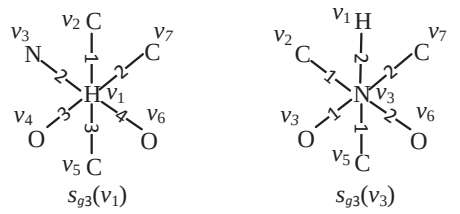
A subgraph pattern f is frequent if its frequency is greater than a pre-defined threshold σ. Since the number of FGGs can be too large and indexing a large number of FGGs will increase the index size and hence the search time, we apply a maximum pattern size threshold, γ, and a maximum edge weight threshold, ρ, to obtain only FGGs with size at most γ and any edge weight at most ρ. In our experiments, we set these thresholds as the best possible values such that the FGGs can fit in the machine memory.

4.2.1 Index Construction

The FP-index consists of two parts: *frequent pattern graph index* (*FPG-index*) and *edge index* (*E-index*). FPG-index stores the FGGs in a B+-tree, with the key as an FGG and the data value as the set of data graphs containing the FGG. To answer queries that may contain infrequent edges, we also construct E-index that builds a B+-tree on the set of infrequent edges and frequent large-weight edges whose weights are larger than ρ, with the key as an edge and the data value as the set of data graphs containing the edge.

4.2.2 Query Processing

Given a query graph q, we process the query with FP-index as follows:



$s_{g3}(v_1)$ $s_{g3}(v_3)$

Figure 2: Two star structures: $s_{g_3}(v_1)$ and $s_{g_3}(v_3)$

- Case 1: q is an FGG indexed by FPG-index. In this case, we obtain q's answer set from the index directly without verification.
- Case 2: q is not an FGG indexed by FPG-index. If q contains infrequent edges, then the candidate set of q is relatively small (at most $\sigma|\mathcal{G}|$), which can be retrieved from E-index. Otherwise, we reduce the weight of some edges in q to obtain an FGG q', then we have a partial answer, $\mathcal{A}(q')$, of q, without verification. And then, we obtain the rest of the answer, i.e., $\mathcal{A}(q) \setminus \mathcal{A}(q')$, as follows. We decompose q into several FGGs in FPG-index and frequent large-weight edges in E-index, as $f_1, f_2, \ldots, f_k$, and compute the candidate set by intersecting the answer sets of these frequent patterns: $\cap_{1\le i\le k}\mathcal{A}(f_i)$.

4.2.3 Complexity Analysis

Similar to other structural graph indexes such as FG-index [4], the construction cost of FP-index is dominated by the cost of mining FGGs and the index size is dominated by the overall size of FGGs. Likewise, the query processing complexity also heavily depends on the number of FGGs indexed as well as the value of $\sigma|\mathcal{G}|$. However, the complexity of mining FGGs, as well as the size and number of FGGs, may vary significantly from database to database and we are not aware of any formal analysis for these factors in the literature.

4.3 Star Index

Since the number of FGGs can be large, FP-index can only be used to process queries of small size efficiently. Moreover, mining FGGs may also be too expensive. Thus, we propose another index, called *Star Index* (*S-index*), which uses only star structures (instead of subgraph structures) to reduce both the index construction and storage overhead, while still capturing much of the structural information for effective filtering in query processing.

For a vertex v in a data graph $g = (V_g, E_g, l_g) \in \mathcal{G}$, we define the *star structure* of v as $s_g(v) = (V_g, E_g^v, l_g, w)$, where (1) v is the center of the star structure; (2) E_g^v consists of the edges from v to other vertices in V_g, that is $E_g^v = \{v\} \times (V_g \setminus \{v\})$; (3) w is a function that assigns the distance $dist_g(v, u)$ to each edge $(v, u) \in E_g^v$, i.e., $w(v, u) = dist_g(v, u)$. For example, Figure 2 shows the star structures of v_1 and v_3 of the graph g_3 in Figure 1.

We group the weights of the edges in a star structure by the label of the non-center end vertex. Let $L = \{l_g(u) : u \in V_g \setminus \{v\}\}$. We obtain a *multiset* of weights (called *weight multiset*) for each label as follows

$$\mathcal{W}_g(v, l) = \{w(u, v) : l_g(u) = l, l \in L, \text{ and } u \in V_g \setminus \{v\}\}.$$

The weight values in the weight multiset are sorted in ascending order. Table 2 lists all the weight multisets for

Table 2: Weight multisets of g_3

Vertex v	$l_{g_3}(v)$	N	C	O	H
v_1	H	2	1,2,3	3,4	
v_2	C	1	1,2	2,3	1
v_3	N		1,1,2	1,2	2
v_4	O	1	2,2,3	1	3
v_5	C	1	2,3	1,2	3
v_6	O	2	1,3,4	1	4
v_7	C	2	1,3	3,4	2

Table 3: Compressed weight multisets of g_3

Label	N	C	O	H
N		1,1,2	1,2	2
C	1	1,2	1,2	1
O	1	1,2,3	1	3
H	2	1,2,3	3,4	

each label and each star structure of g_3. For example, for the star structure $s_{g_3}(v_1)$, $\mathcal{W}_{g_3}(v_1, \mathtt{C}) = \{1, 2, 3\}$, $\mathcal{W}_{g_3}(v_1, \mathtt{O}) = \{3, 4\}$, and $\mathcal{W}_{g_3}(v_1, \mathtt{N}) = \{2\}$.

Given two weight multisets $\mathcal{W}_1 = \{w_1, \ldots, w_k\}$ and $\mathcal{W}_2 = \{w'_1, \ldots, w'_t\}$, where $k \leq t$. We define the merge of $\mathcal{W}_1$ and $\mathcal{W}_2$ as follows.

$$\mathcal{W}_1 \cap \mathcal{W}_2 = \{\min(w_1, w'_1), \ldots, \min(w_k, w'_k), w'_{k+1}, \ldots, w'_t\}.$$

For some vertices in a graph, they may share the same label. So, we further compress the weight multisets as follows.

$$\mathcal{W}_g(l_1, l_2) = \cap_{l_g(v)=l_1, v \in V_g} \mathcal{W}_g(v, l_2).$$

Table 3 lists the compressed weight multisets. For example, v_2 and v_5 share the same label C, so $\mathcal{W}_{g_3}(v_2, \mathtt{H}) = \{1\}$ and $\mathcal{W}_{g_3}(v_5, \mathtt{H}) = \{3\}$ are merged into one $\mathcal{W}_{g_3}(\mathtt{C}, \mathtt{H}) = \{\min(1, 3)\} = \{1\}$.

The set of distinct compressed weight multisets of each label pair (l_1, l_2) in the database $\mathcal{G}$ can be obtained as follows.

$$\mathcal{W}(l_1, l_2) = \cup_{g \in \mathcal{G}} \mathcal{W}_g(l_1, l_2).$$

For each compressed weight multiset $w \in \mathcal{W}(l_1, l_2)$, the set of data graphs whose corresponding compressed weight multiset is w is defined as follows.

$$\mathcal{A}(w) = \{g : \mathcal{W}_g(l_1, l_2) = w \text{ and } g \in \mathcal{G}\}.$$

4.3.1 Index Structure

S-index is constructed based on $\mathcal{W}(l_1, l_2)$ for each distinct label pair (l_1, l_2) and $\mathcal{A}(w)$ for each $w \in \mathcal{W}(l_1, l_2)$. The index structure of S-index consists of the following parts:

- A B+-tree on the set of distinct label pairs with each pair (l_1, l_2) as the key and $\mathcal{W}(l_1, l_2)$ as the data value. We name this B+-tree as SI.
- A nested B+-tree on the set of distinct compressed weight multisets $\mathcal{W}(l_1, l_2)$ for each pair (l_1, l_2), where each compressed weight multiset $w \in \mathcal{W}(l_1, l_2)$ is the key and $\mathcal{A}(w)$ is the data value. We name this nested B+-tree as $SI(l_1, l_2)$.

Algorithm 3 outlines the construction of S-index. For each data graph $g \in \mathcal{G}$, for each distinct label pair (l_1, l_2) in g, we obtain the compressed weight multiset $\mathcal{W}_g(l_1, l_2)$, and store it in the record with the key (l_1, l_2) in SI. Then, we access the record with the key $\mathcal{W}_g(l_1, l_2)$ in the nested B+-tree $SI(l_1, l_2)$ and add g to the corresponding record.

Algorithm 3: BUILD-SINDEX($\mathcal{G}$)

input : the graph database, $\mathcal{G}$
output: the S-index, SI

```
for g ∈ G do
    Let L(g) = {(l_g(u), l_g(v)) : u, v ∈ V_g and u ≠ v} ;
    for (l_1, l_2) ∈ L(g) do
        Compute w = W_g(l_1, l_2) ;
        Add w to the record in SI with key (l_1, l_2) ;
        Add g to the record in SI(l_1, l_2) with key w ;
return SI
```

Algorithm 4: QUERY-SINDEX($q, SI, \mathcal{G}$)

input : the query graph, $q = (V_q, E_q, l_q, t)$
S-Index, SI
the graph database $\mathcal{G}$
output: the candidate set of q, $\mathcal{C}(q)$

```
C(q) ← G ;
Let L(q) = {(l_g(u), l_g(v)) : u, v ∈ V_g and u ≠ v} ;
for (l_1, l_2) ∈ L(q) do
    Compute W_q(l_1, l_2) ;
    C(l_1, l_2) ← ∅ ;
    for w ∈ SI(l_1, l_2) and w ≤ W_q(l_1, l_2) do
        C(l_1, l_2) ← C(l_1, l_2) ∪ A(w) ;
    C(q) ← C(q) ∩ C(l_1, l_2) ;
return C(q)
```

4.3.2 Query Processing

We now discuss query processing by S-index. Given two compressed weight multisets $\mathcal{W}_1 = \{w_1, \ldots, w_k\}$ and $\mathcal{W}_2 = \{w'_1, \ldots, w'_t\}$, we say that $\mathcal{W}_1 \leq \mathcal{W}_2$ if and only if 1) $k \geq t$, and 2) $w_r \leq w'_r$ for $1 \leq r \leq t$.

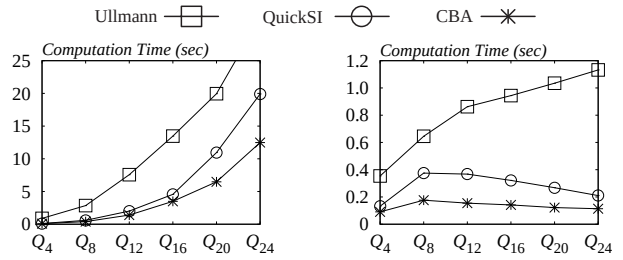
LEMMA 2. *Let $s_g(u)$ and $s_q(v)$ be two star structures, such that u and v are vertices in graphs $g = (V_g, E_g, l_g)$ and $q = (V_q, E_q, l_q, t)$, respectively. If $s_g(u)$ matches $s_q(v)$, then we have $\mathcal{W}_g(l_1, l_2) \leq \mathcal{W}_q(l_1, l_2)$ for all distinct label pairs (l_1, l_2) of q.*

PROOF. Consider a label pair (l_1, l_2) of q, l_1 is the label of the center vertex of a star structure of q and l_2 is the label of a non-center vertex. Since g matches q, the number of vertices in V_g with label l_2 is no smaller than the number of vertices in V_q with the same label. Therefore, we have $|\mathcal{W}_g(l_1, l_2)| \geq |\mathcal{W}_q(l_1, l_2)|$. Moreover, for each vertex $v' \in V_q$ of label l_2, there exists a vertex $u' \in V_g$ such that u' maps to v' and $dist_g(u, u') \leq dist_q(v, v')$. Thus, the proof is complete. □

According to Lemma 2, we process a query by S-index as shown in Algorithm 4. For each distinct label pair (l_1, l_2) of q, we merge all the candidate sets associated with (l_1, l_2) that are smaller than $\mathcal{W}_q(l_1, l_2)$, which gives $\mathcal{C}(l_1, l_2)$. The candidate set of q is then obtained by intersecting $\mathcal{C}(l_1, l_2)$ for all distinct pairs (l_1, l_2) of q.

4.3.3 Complexity Analysis

Let m be the average number of distinct labels in a data graph. Thus, the number of distinct label pairs is $O(m^2)$, and the number of compressed weight multisets in



(a) Average Edge Weight = 1.5 (b) Average Edge Weight = 1

Figure 3: Efficiency of Generalized Subgraph Isomorphism Algorithms

the database $\mathcal{G}$ is $O(m^2|\mathcal{G}|)$. Assume that the average number of vertices in a data graph is α, then the average size of a compressed weight multiset is α/m. Thus, the space complexity of S-index is $O(\alpha m|\mathcal{G}|)$.

Assume that the number of distinct labels in a query graph q is t. The index response time is the summation of the time for searching the compressed weight multisets of q, thus the running time complexity is $O(t^2 \log(m^2|\mathcal{G}|))$. Note that, in real life queries, the number of distinct labels is often small.

5. EXPERIMENTAL RESULTS

This section experimentally evaluates our indices and algorithms for generalized subgraph matching. Section 5.1 describes the experimental settings. Section 5.2 evaluates our algorithms for the generalized subgraph isomorphism problem, and Section 5.3 tunes the parameters for the proposed FP-Index. After that, Sections 5.4 and 5.5 demonstrate the efficiency of our indexing methods on real and synthetic datasets, respectively.

5.1 Experimental Settings

Datasets. We use two benchmark datasets commonly adopted in the literature [8, 18]. Both datasets contain graphs that represent chemical molecules. The first one is the AIDS Antiviral Screen Dataset [18], which consists of 10,000 graphs. The second dataset is referred to as PubChem [18], and it contains 100,000 graphs. We use PubChem.mK to denote a sample set of PubChem with m thousands of graphs. In addition, we use synthetic datasets produced from GraphGen[2], a public available synthetic graph generator.

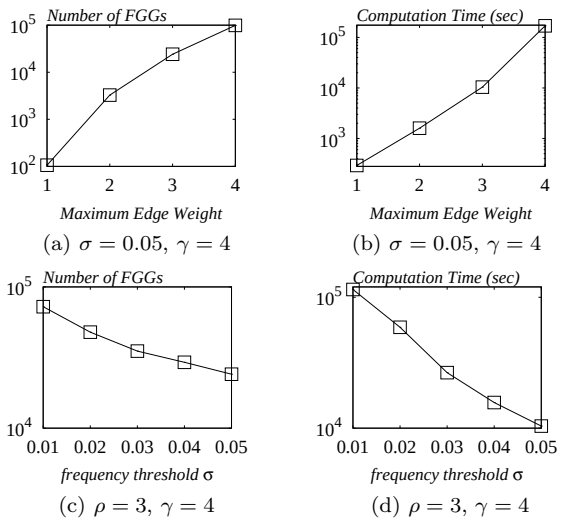
Query sets. For the AIDS dataset, we adopt the query sets from [18], but we ignore the label on each edge and add a weight on the edge (since we target at query graphs where the edges are unlabelled and weighted). For the other datasets, we generate the query sets by first extracting generalized subgraphs from the graphs in the datasets, such that the number of data graphs matching each extracted generalized subgraph is at most 10% of the total number of data graphs. In other words, we avoid generating generalized subgraph matching queries that would return excessive numbers of results.

All of our experiments are conducted on a machine with a Intel Xeon 2.4GHz CPU with 48GB RAM.

5.2 Generalized Subgraph Isomorphism

Our first set of experiments compares three algorithms for generalized subgraph isomorphism: our cost-based approach (denoted as CBA), as well as the extensions of Ullmann's algorithm and QuickSI. Figure 3 illustrates the average running time required by each algorithm to match each query graphs in query set Q_i to all data graphs in the AIDS dataset. In particular, each query set Q_i contains 1000 query graphs, and each query graph in Q_i contains i vertices. Figure 3a shows the results when the edges in the query graph have average weight 1.5. Observe that CBA considerably outperforms the extension of QuickSI, which in turn is superior than the extension of Ullman's algorithm. Figure 3b shows the results when the average edge weight in the query graph equals 1, i.e., when the generalized subgraph isomorphism problem degenerates to the standard subgraph isomorphism problem. Even in this degenerated case, CBA still consistently outperforms QuickSI and Ullmann's algorithm. This demonstrates the superiority of our cost-based method for optimizing vertex matching order. We have conducted a similar set of experiments on the PubChem datasets, and we found that the results are qualitative similar; we omit those results for the interests of space.

[2]http://www.cse.ust.hk/graphgen/



(a) $\sigma = 0.05$, $\gamma = 4$ (b) $\sigma = 0.05$, $\gamma = 4$

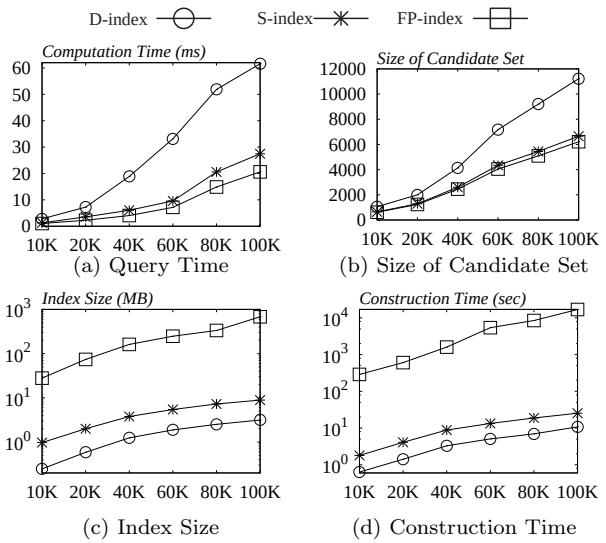
(c) $\rho = 3$, $\gamma = 4$ (d) $\rho = 3$, $\gamma = 4$

Figure 4: Space and Pre-computation Costs of the FP-index

5.3 Tuning the FP-index

The second set of our experiments evaluation the space and pre-computation costs of the FP-index (presented in Section 4.2) on a set of 40 thousands data graphs sampled from the PubChem dataset. Figure 4a illustrates the number of Frequent Generalized subGraphs (FGG) that need to be stored in the FP-index, varying the maximum edge weight ρ in the graph patterns from 1 to 4, with the frequency threshold set to $\sigma = 0.05$ and the maximum number of vertices in the FGGs set to $\gamma = 4$. Note that the number of FGGs increases exponentially with maximum edge weight ρ. Figure 4b shows the time required to mine the FGGs, which also exhibits an exponential growth with the increase of ρ. These results indicate that maximum edge weight adopted in the construction in the FP-index have to be carefully selected and has to be reasonably small.

Figures 4c and 4d illustrate the number of FGGs and pre-computation time required by FP-index, respectively, varying the frequency threshold σ from 0.01 to 0.05, with the maximum edge weight set to $\rho = 3$ and the maximum number of vertices in the FGGs set to $\gamma = 4$. Observe that



(a) Query Time (b) Size of Candidate Set

(c) Index Size (d) Construction Time

Figure 5: Index Performance v.s. Dataset Size

both the number of FGGs and the pre-computation time decreases exponentially when the frequency threshold σ increases. Hence, we may use a large σ to reduce the space and pre-computation cost of the FP-index. One may be tempted to set σ to be even larger than 0.05, which, however, may significantly reduce the effectiveness of FP-index, as an excessively large σ would make it difficult for FP-index to answer a query without invoking the verification process.

Based on the results in Figure 4, we set $\rho = 3$, $\sigma = 0.05$, and $\gamma = 4$ for the FP-index in all following experiments.

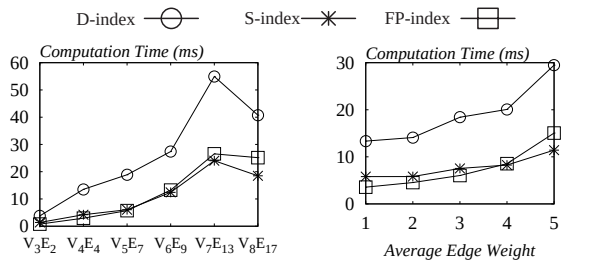
5.4 Performance of Indices

Our next set of experiments compares the performance of the three proposed indices (i.e., D-index, S-index, and FP-index) in terms of query processing performance, space overhead, and construction time. For these experiments, we use sample sets of PubChem dataset with sizes varying from 10K to 100K. We do not use the AIDS dataset as it contains only a small number of data graphs.

Figure 5a illustrates average query processing time of each index for a query set with 1000 graphs, such that on average each graph has 5 vertices and 7 edges, and the average edge weight equals 2.5. Both the S-index and the FP-index significantly outperforms the D-index, and the FP-index is slightly better than the S-index. This is consistent with the results in Figure 5b, which shows the average size of the candidate set induced by each index during query processing. As shown in Figures 5c and 5d, however, the space and construction overheads of FP-index are significantly higher than those of the S-index, which in turn are higher than those of the D-index.

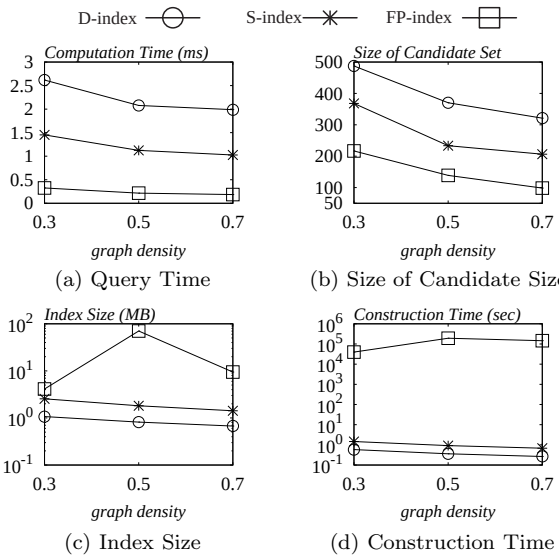
Figure 6a shows the average query time of each index for query sets V_iE_j on the dataset with 40K data graphs, such that each query set V_iE_j contains 1000 query graphs, each of which has i vertices and j edges, and the average weight of the edges equals 2.5. The FP-index achieves the smallest query time when the numbers of vertices and edges in the query graphs are small, but it is outperformed by the S-index on large query graphs. In addition, the D-index is consistently slower than both the FP-index and the S-index.

Figure 6b illustrates the average query time of each index for query set V_5E_7, with the average edge weight varying



(a) Varying Query Graph Size (b) Varying Max. Edge Weight

Figure 6: Index Performance v.s. Parameters of the Query Graphs



(a) Query Time (b) Size of Candidate Size

(c) Index Size (d) Construction Time

Figure 7: Index Performance v.s. Graph Density

from 1 to 5. The FP-index performs the best when the average edge weight is no more than 3, which is the maximum edge weight handled in its preprocessing step. When the average edge weight is larger than 3, however, the performance of the FP-index degrades, and the S-index becomes the most efficient one.

5.5 Performance on Synthetic Dataset

The experiments use synthetic datasets to evaluate the performance of our indices with respect to a parameter that has not been investigated in the previous experiments, i.e., the densities of the data graphs. In particular, the density of a data graph with n vertices and m edges equals $m/\binom{n}{2}$. We generate synthetic graphs with densities varying from 0.3 to 0.7, and we use them to construct datasets, such that each dataset contains 10K data graphs, each of which has 30 edges and a fixed density. The query graphs for each dataset is constructed in a manner similar to previous experiments, such that each query graph on average has 5 vertices, 7 edges, with an average edge weight 2.5.

Figure 7 illustrates the performance of each index as a function of the data graph density. As with our previous experiments, the FP-index achieves the best query performance, but it incurs the highest space and pre-computation overheads. The D-index requires the smallest space and preprocessing time, but its query time is the largest. The S-index consistently lands on the middle ground between the FP-index and the D-index.

Summary. Our experiments show that the FP-index offers superior query performance at the cost of space and pre-computation time. Therefore, it is suitable for the applications where (i) efficient query processing is crucial, and (ii) space and pre-computation overheads are not a major concern. In contrast, the D-index entails relatively high query cost, but it incurs minimal space and preprocessing overhead. This renders it preferable in the scenarios with stringent requirements on space consumption or pre-computation time. Finally, the S-index's space and pre-computation costs are only slightly higher than that of the D-index, but its query efficiency is almost comparable to that of the FP-index. Hence, it offers user a choice to strike a good balance between query processing and space (preprocessing) overheads.

6. RELATED WORK

There are some existing studies of graph matching problem by allowing edges to map to paths for graphs [6,7,10,25]. However, their queries are too rigid by fixing the length of all the mapping paths [25], or too relax by allowing node similarity matching [7]. Besides, all these works are tailored to query a single large graph making them unsuitable for querying a large set of small or medium-sized graphs.

On the other hand, various types of graph query processing on a large set of small or medium-sized graphs have been studied in the literature in recent years and we restrict our discussion on the closely related ones, namely subgraph query processing [4, 9, 13, 15, 18, 21, 23], supergraph query processing [2, 3, 14, 22], and similarity graph query processing [12, 19, 20, 24]. All these works proposed some indexing techniques to filter out as many unmatching data graphs as possible. Although many different types of graph indexing techniques have been proposed, none of them is similar to our indexes except FG-index [4], which is similar to FP-index. However, the only similarity lies on the use of frequent patterns to avoid verification and the use of infrequent edges to reduce the candidate set size, while the index structure of FP-index (which builds on B+-trees) is totally different from that of FG-index (which is an unbalanced tree built on the clusters of frequent patterns). Apart from that, both D-index and S-index are entirely different from all existing indexes. In addition, our work is the first to propose indexes for processing generalized subgraph queries.

7. CONCLUSIONS

We studied a new type of graph queries, generalized subgraph queries. We proposed a succinct and effective cost model to minimize the cost of generalized subgraph isomorphism. We also developed three indexes that can effectively filter out unmatching data graphs, which significantly reduces the total query response time. We evaluated our algorithms with experiments on both real datasets and synthetic datasets. The results show that our matching algorithm is efficient in candidate verification as it considerably outperforms the direct extension of existing graph matching algorithms, while our indexes are also effective in filtering. Thus, the results verify that our method is efficient in query processing (in both filtering and candidate verification). Although some of the indexes have weaknesses, we show how the weaknesses are addressed by another index; in particular, our results show that S-index achieves both a low index construction cost and a short query response time.

8. ACKNOWLEDGMENTS

Xiaokui Xiao was supported by Nanyang Technological University under SUG Grant M58020016 and AcRF Tier 1 Grant RG 35/09, and by the A*STAR SERG Grants 1021580074. James Cheng was supported in part by the A*STAR TSRP Grants 1021580034 and 1121720013.

9. REFERENCES

[1] C. C. Aggarwal and H. Wang, editors. *Managing and Mining Graph Data*, volume 40 of *Advances in Database Systems*. Springer, 2010.

[2] C. Chen, X. Yan, P. S. Yu, J. Han, D.-Q. Zhang, and X. Gu. Towards graph containment search and indexing. In *VLDB*, pages 926–937, 2007.

[3] J. Cheng, Y. Ke, A. W.-C. Fu, and J. X. Yu. Fast graph query processing with a low-cost index. *VLDB J.*, 20(4):521–539, 2011.

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

[5] S. A. Cook. The complexity of theorem-proving procedures. In *STOC*, pages 151–158, 1971.

[6] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu. Adding regular expressions to graph reachability and pattern queries. In *ICDE*, pages 39–50, 2011.

[7] W. Fan, J. Li, S. Ma, H. Wang, and Y. Wu. Graph homomorphism revisited for graph matching. *PVLDB*, 3(1):1161–1172, 2010.

[8] W.-S. Han, J. Lee, M.-D. Pham, and J. X. Yu. igraph: A framework for comparisons of disk-based graph indexing techniques. *PVLDB*, 3(1):449–459, 2010.

[9] H. He and A. K. Singh. Closure-tree: An index structure for graph queries. In *ICDE*, page 38, 2006.

[10] W. Jin and J. Yang. A flexible graph pattern matching framework via indexing. In *SSDBM*, pages 293–311, 2011.

[11] C. L. S. John W. Moore and P. C. Jurs. *Chemistry: The Molecular Science*, volume 2. Brooks Cole, 2007.

[12] H. Shang, X. Lin, Y. Zhang, J. X. Yu, and W. Wang. Connected substructure similarity search. In *SIGMOD*, pages 903–914, 2010.

[13] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: An efficient algorithm for testing subgraph isomorphism. *In VLDB*, 2008.

[14] H. Shang, K. Zhu, X. Lin, Y. Zhang, and R. Ichise. Similarity search on supergraph containment. In *ICDE*, pages 637–648, 2010.

[15] D. Shasha, J. T.-L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *PODS*, pages 39–52, 2002.

[16] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[17] J. H. Van Drie, D. Weininger, and Y. C. Martin. Aladdin: An integrated tool for computer-assisted molecular design and pharmacophore recognition from geometric, steric, and substructure searching of three-dimensional molecular structures. *Journal of Computer-Aided Molecular Design*, 3:225–251, 1989.

[18] X. Yan, P. S. Yu, and J. Han. Graph indexing: A frequent structure-based approach. In *SIGMOD*, pages 335–346, 2004.

[19] X. Yan, P. S. Yu, and J. Han. Substructure similarity search in graph databases. In *SIGMOD*, pages 766–777, 2005.

[20] Z. Zeng, A. K. H. Tung, J. Wang, J. Feng, and L. Zhou. Comparing stars: On approximating graph edit distance. *PVLDB*, 2(1):25–36, 2009.

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

[22] S. Zhang, J. Li, H. Gao, and Z. Zou. A novel approach for efficient supergraph query processing on graph databases. In *EDBT*, pages 204–215, 2009.

[23] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: Tree + delta >= graph. In *VLDB*, pages 938–949, 2007.

[24] X. Zhao, C. Xiao, X. Lin, and W. Wang. Efficient graph similarity joins with edit distance constraints. In *ICDE*, pages 834–845, 2012.

[25] L. Zou, L. Chen, and M. T. Özsu. Distancejoin: Pattern match query in a large graph database. *PVLDB*, 2(1):886–897, 2009.