Fast Subgraph Search with Graph Code Indices

Naoya Funamoto and Akihiro Inokuchi(✉)

Graduate School of Science and Technology, Kwansei Gakuin University, Sanda, Hyogo, Japan
{ihn04482,inokuchi}@kwansei.ac.jp

Abstract. The subgraph search problem is of fundamental importance in the fields of information science and database management. In this paper, we propose an index-based subgraph search method that is as fast as the current state-of-the-art technique. The proposed method is an extension of CodeTree, which is a supergraph search method that uses neither enumeration nor graph mining. The extended CodeTree$_{sub}$ treats graphs as graph codes and uses the prefix tree for these graph codes as an index. This index permits the highly efficient filtering of non-solutions, but its construction entails little computational overhead. CodeTree$_{sub}$ effectively limits the number of candidate solutions so that only induced subgraphs of graphs in databases are included in the index, thus accelerating the filtering step. Additionally, CodeTree$_{sub}$ can identify some solutions during the filtering stage. The result is a scalable, high-speed graph filtering and verification method. We compared the performance of CodeTree$_{sub}$ with that of two non-index-based techniques on six benchmark datasets. The results demonstrated that the proposed method was consistently as fast as or faster than the state-of-the-art VEQ$_S$ method in terms of query processing. This study is of particular interest because it illustrates that index-based methods have the potential to outperform non-index-based techniques, thereby providing enhanced query speeds for small- and large-scale databases alike.

1 Introduction

A graph is a data structure that represents objects and the relationships among them. For example, atoms and chemical bonds in molecules may correspond to vertices and edges in graphs, respectively, which allows molecules to be represented as graphs. Additionally, when proteins and the interactions among them correspond to vertices and edges, protein–protein interactions can be represented as graphs. Many objects can be represented in the form of graphs, such as human relational networks, hyperlink structures, and function calls in computer programs. When such objects are represented by graphs and stored in databases, searching for some desired graphs becomes an essential technology in the field of information science. Various graph search techniques exist, such as finding a graph with a perfect match [5,6], a graph with a matching substructure [12,18],

C. Strauss et al. (Eds.): DEXA 2024, LNCS 14910, pp. 43–58, 2024.
https://doi.org/10.1007/978-3-031-68309-1_4

or a graph with a similar structure [16,25]. The subgraph isomorphism problem is NP-complete; hence, solving the subgraph search problem requires the design of efficient algorithms.

Algorithm 1: IFV Procedure

Input : query graph q and index $\mathcal{I}$
Output: solutions $S = \{g_i \in G \mid q \text{ is a subgraph of } g_i\}$

1 $P(q) \leftarrow$ Decompose q into a set of patterns
2 $Can \leftarrow \bigcap_{p \in P(q)} lookup(\mathcal{I}, p)$
3 $S \leftarrow \emptyset$
4 **for** $g \in Can$ **do**
5 **if** $verification(g, q)$=*true* **then**
6 $S \leftarrow S \cup \{g\}$
7 **return** S

Algorithm 2: vcFV Procedure

Input : set of graphs $G = \{g_1, g_2, \ldots, g_n\}$ and query graph q
Output: solutions $S = \{g_i \in G \mid q \text{ is a subgraph of } g_i\}$

1 $S \leftarrow \emptyset$
2 **for** $g_i \in G$ **do**
3 $\mathcal{A} \leftarrow filter(g_i, q)$
4 **if** $\mathcal{A} \neq null$ **then**
5 **if** $verification(g_i, q, \mathcal{A})$= *true* **then**
6 $S \leftarrow S \cup \{g_i\}$
7 **return** S

Algorithm 1 outlines the typical indexing–filtering–verification (IFV) method [18] for the subgraph search problem. In advance of receiving queries, IFV constructs an index $\mathcal{I}$ for a set of graphs G using an enumeration technique. Given a query q, IFV decomposes q into a set of patterns $P(q)$ and obtains $G_S(p) = \{g_i \in G \mid p \text{ is a subgraph of } g_i\}$ with $lookup(\mathcal{I}, p)$ for each pattern $p \in P(q)$. The intersection of sets $G_S(p)$ contains the candidate solutions $Can \subseteq G$. For each candidate $g \in Can$ and q, IFV verifies the subgraph isomorphism problem on Line 5.

Although various IFV-based methods for solving the problem have been proposed since 2000, current mainstream methods are index-free techniques such as CFQL [18] and VEQ_S. In the paper in which VEQ_S was proposed [12], the authors state the following:

> Based on our empirical study, building an existing index and filtering using the index incur considerable overhead without gaining higher filtering power for most queries, which is already confirmed by [18]; indeed, the state-of-the-art subgraph search algorithm CFQL [18] has shown that existing indexing methods followed by recent preprocessing and enumeration techniques are inefficient in query processing on widely-used datasets such as PDBS, PCM, and PPI.

CFQL and VEQ_S are based on the vertex connectivity-based filtering–verification (vcFV) framework, the outline of which is presented in Algorithm 2. Given a query q, vcFV constructs an auxiliary data structure $\mathcal{A}$ for q and each graph $g_i \in G$. If $\mathcal{A}$ is not constructed, g_i cannot be a solution; otherwise, vcFV

solves the subgraph isomorphism problem for g_i and q with $\mathcal{A}$. Unlike Algorithm 1, in Algorithm 2, the *verification* step uses $\mathcal{A}$, which greatly reduces the search space for the subgraph isomorphism between q and g_i.

Although the use of indices has been discouraged as a tool for subgraph search methods in recent years, there are advantages to using indices in database searches. For example, in relational databases, balance trees are often used as indices. The use of these trees reduces the computational complexity of searches to $\mathcal{O}(\log n)$, where n is the number of tuples in a database. By contrast, the computational complexity of Algorithm 2 is proportional to the number of graphs. Therefore, it is necessary to review the use of indices in graph databases. In this paper, we propose an index-based method for the subgraph search problem. The characteristics of the proposed CodeTree$_{sub}$ method are as follows:

1. non-solution graphs are filtered with high efficiency using indices,
2. indices are constructed without considerable overheads, and
3. the high speed and high filtering performance result in short search times.

2 Preliminaries

A labeled graph is represented as $g = (V, E, \ell)$, where V is a set of vertices, $E \subseteq V \times V$ is a set of edges, and $\ell : V \cup E \to \Sigma$ is a function for assigning labels Σ to the vertices and edges. In this paper, we express the vertices and edges of g as $V(g)$ and $E(g)$, respectively. Given two graphs $g = (V, E, \ell)$ and $g' = (V', E', \ell')$, if there is an injective function $\phi : V \to V'$ that satisfies $\forall v, u \in V$, then g is called a subgraph of g', which is denoted by $g \sqsubseteq g'$:

- $\ell(v) = \ell'(\phi(v))$
- $(\phi(v), \phi(u)) \in E'$ if $(v, u) \in E$
- $\ell((v, u)) = \ell'((\phi(v), \phi(u)))$.

Additionally, if $(\phi(v), \phi(u)) \in E'$ iff $(v, u) \in E$ is also satisfied, then g is called an induced subgraph of g', which is denoted by $g \sqsubseteq_i g'$. The problem of whether $g \sqsubseteq g'$ is called the subgraph isomorphism problem. This problem is NP-complete.

Given graphs $G = \{g_1, g_2, \ldots, , g_n\}$ and query graph q as input, the problem we address in this paper is to output a set of solutions $S = \{g_i \in G \mid q \sqsubseteq g_i\}$.

We propose a method based on IFV. The basic idea of IFV-based methods is that, if $p \sqsubseteq q \wedge p \not\sqsubseteq g_i$, then $q \not\sqsubseteq g_i$ [23]. To use this property, IFV computes whether $p \sqsubseteq g_i$ for various patterns p in advance and stores $G_S(p) = \{g_i \in G \mid p \sqsubseteq g_i\}$ for each p. Then, by computing $Can = \bigcap_{p \in P} G_S(p)$ for a set of patterns P, each of which is a subgraph of q, the graphs in G that are not solutions are filtered out, and a set of candidates $Can \subseteq G$ is obtained. Finally, the subgraph isomorphism problem between $g \in Can$ and q is solved. The index $\mathcal{I}$ holds the set of patterns and $G_S(p)$ for each pattern p. In this paper, we discuss the design of such an index.

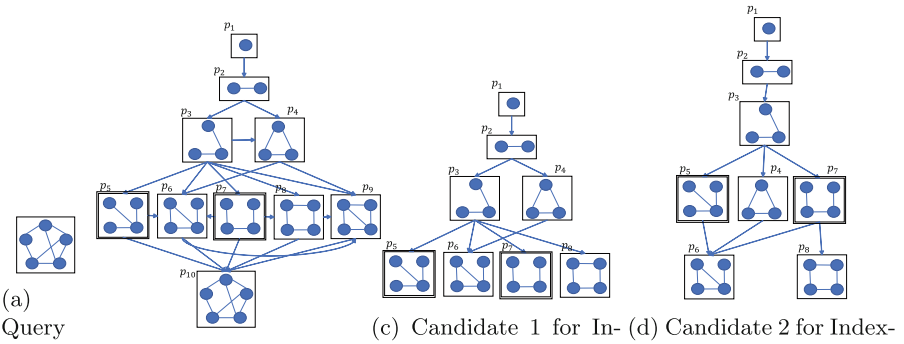
3 Related Work

Methods based on enumeration techniques [1,7,13,15,17,19] exhaustively enumerate all possible patterns in graphs G and store them in an index. The variety of patterns is huge; hence, the size of the index becomes enormous and significant amounts of memory space are required to construct the index. Therefore, this type of method generally limits the number of patterns to simple structures such as paths, cycles, and trees. For example, GraphGrep [17], GraphGrepSX (GGSX) [1], GRAPES [7], and SING [15] enumerate paths from graphs G, whereas CT-Index [13] enumerates cycles. The number of patterns is restricted by limiting the number of vertices or edges within each pattern.

Methods based on graph mining search for subgraph patterns that occur frequently in G, and construct indices from these mined patterns [3,4,20,23,24]. The support of each pattern p is defined as $sup(p) = |\{g_i \in G \mid p \sqsubseteq g_i\}|$. For a given threshold σ, frequent subgraph patterns in G are $\{p \mid sup(p) \geq \sigma\}$ [10]. In addition to the support, other methods exist for selecting patterns by measuring the filtering ability of the patterns. For example, methyl groups and benzene rings are present in many organic compounds, so they are not always suitable for proper filtering. gIndex [22] uses the discriminative ratio for selection. Mining-based methods require thresholds to be applied to the support or discriminative ratio. It is sometimes difficult to adjust these thresholds, which makes it necessary to repeat the index construction process when they are changed, which entails a long computation time.

Methods based on enumeration and mining are time-consuming for index construction, which makes them ineffective for filtering. Hence, methods based on vcFV without indexing have been proposed in recent years. CFQL [18] constructs an auxiliary data structure called the compact path-index (CPI) between q and $g_i \in G$ during the prepossessing stage, and then performs a verification step with GraphQL [8]. CPI is a spanning tree of the query graph, and each node in the tree has candidate vertices in g_i that may correspond to the node. CPI removes false positive candidates for the node of the query graph and can also determine the most efficient matching order between the vertices in two graphs. By contrast, VEQ_S [12] searches for matching between two graphs. It generates more compact auxiliary data structures between q and g_i than CPI. In this process, the search space is reduced by skillfully handling the neighbor equivalence class among all degree-one vertices in q. Additionally, VEQ_S checks whether two children of each node in the search tree are equivalent using this data structure and prunes the redundant search subspace. The index and vertex connectivity-based filtering–verification framework [18] performs a subgraph search by applying vcFV after filtering non-solutions with indices.

Although we address the subgraph search problem in this paper, we should also discuss the related supergraph search problem, which attempts to find $\{g_i \in G \mid g_i \sqsubseteq q\}$ for some given G and q, [2,9,11,14]. Methods based on indices with enumeration and mining techniques form the bulk of supergraph search techniques, although index-free methods have been proposed recently [11]. Additionally, CFQL and VEQ_S can solve the supergraph search problem by replacing q and g_i on Lines 3 and 5 in Algorithm 2.



(a) Query Graph q (b) All Subgraphs of q (c) Candidate 1 for Indexing (d) Candidate 2 for Indexing

Fig. 1. Relationships between the Query Graph and its Connected Subgraphs

4 Basic Concept of the Proposed Method

Next, we explain the basic concept of indexing in the proposed method. Figure 1(a) shows an example of query graph q. All connected subgraphs $P_c = \{p_1, p_2, \ldots, p_{10}\}$ of g are depicted in Fig. 1(b). The graphs are connected by arrows from p_i to p_j in one of the following cases:

- $p_i \sqsubset p_j \wedge |V(p_i)| + 1 = |V(p_j)|$ or
- $p_i \sqsubset p_j \wedge |E(p_i)| + 1 = |E(p_j)|$.

After decomposing q into subgraphs, we obtain sets of graphs $\{g_i \in G \mid p \sqsubseteq g_i\}$ with $lookup(\mathcal{I}, p)$ in Algorithm 1. By intersecting these sets, we limit the candidate solutions to $\bigcap_{p \in P_c} lookup(\mathcal{I}, p) = \bigcap_{p \in P_c} \{g_i \in G \mid p \sqsubseteq g_i\}$. In the case in which the index stores all possible connected graphs, huge amounts of time and memory are required to construct and hold the index [19]. Therefore, it is not practical to store the graphs in the index. Hence, for example, we assume that p_9 and p_{10} are not stored in the index.

To efficiently traverse an index that does not store all possible graphs, we introduce the following lemma. We omit proofs in this paper because of the space limitation.

Lemma 1. *Given query graph q and two patterns p_i and p_j such that $p_i \sqsubset p_j \sqsubseteq q$, candidate solutions for q are included in*

$$\{g \in G \mid p_i \sqsubseteq g\} \cap \{g \in G \mid p_j \sqsubseteq g\} = \{g \in G \mid p_j \sqsubseteq g\}. \quad \blacksquare \tag{1}$$

According to Lemma 1, applying $lookup(\mathcal{I}, p)$ with a larger pattern results in more effective filtering. The maximum patterns among $P'_c = P \setminus \{p_9, p_{10}\} = \{p_1, p_2, \ldots, p_8\}$ are p_6 and p_8. Patterns p_5 and p_7, enclosed by the double-line square, are not induced subgraphs of q. When p' is not an induced subgraph of q, it is possible that p'', which contains p' as a subgraph and is an induced subgraph of q, will be stored in the index. In this case, according to Lemma 1, p'' is more effective for filtering than p'. When traversing the index for a given q, we must consider how to efficiently reach p_6 and p_8 via the patterns $P'_c \setminus \{p_5, p_7\}$.

We consider two cases and redraw Fig. 1(b). The first case is that $p_i \sqsubset p_j$ for all graphs in P'_c and $|V(p_i)| + 1 = |V(p_j)|$, as shown in the directed acyclic graph (DAG) in Fig. 1(c). The second case is that $p_i \sqsubset p_j$ for all graphs in P'_c and $|E(p_i)| + 1 = |E(p_j)|$, as shown in the DAG in Fig. 1(d). We wish to obtain patterns p such that $p \sqsubseteq q$ by adopting one of the DAGs as an index and traversing the adopted DAG.

Note that the objective is to reach p_6 and p_8 in Fig. 1(c) or 1(d), not to visit all nodes[1]. Therefore, it is desirable to visit fewer nodes so that the index traversal is more efficient. For the case of Fig. 1(c), we can reach p_6 and p_8 without visiting patterns that are not induced subgraphs of q, which enables us to reduce the time required to traverse the index. By contrast, for Fig. 1(d), to reach p_8, it is necessary to pass through node p_7, which is not an induced subgraph of q; this increases the time taken to traverse the index. The gIndex method [22] uses an index that is a spanning tree of the DAG in Fig. 1(d). Patterns found in the nodes in this index are represented as a depth-first search code [21]. Note that gIndex may visit nodes with patterns that are subgraphs of q, but are not induced subgraphs of q. By contrast, we aim to design a method for traversing the DAG in Fig. 1(c) without visiting patterns that are not induced subgraphs of q. For this purpose, we use the Apriori-based connected Graph Mining (AcGM) code [10].

5 Graph Representation and Indexing of Databases

To represent graphs, we use the AcGM code.

Definition 1 (AcGM code [10]). *When the vertex IDs $u_1, u_2, \ldots, u_{|V|}$ are assigned to the vertices in a graph $g = (V, E, \ell)$, the graph is represented as the adjacent matrix, where subgraphs induced by $u_1, u_2, \ldots, u_i$ $(1 \leq i \leq |V|)$ are connected. In the matrix, if $(u, u') \in E$, $x_{u,u'} = \ell((u, u'))$; otherwise, $x_{u,u'} = 0$. In this case, the AcGM code of g is expressed as*

$$code(g, \langle u_1, u_2, \ldots, u_{|V|} \rangle) = s_1 s_2 \cdots s_{|V|},$$

$$where s_i = \ell(u_i) x_{1,i} x_{2,i} \cdots x_{i-1,i}.$$

s_i $(1 \leq i \leq |V|)$ is called a code fragment. ■

For a given graph, multiple AcGM codes exist for the different ways of assigning vertex IDs. We denote a set of AcGM codes that represent g by $\Omega(g)$ and a graph represented by a code c by $g(c)$.

For a given set of AcGM codes, we define its prefix tree as the Code Tree.

Definition 2 (Code Tree [9]). *The Code Tree consists of a triplet (N, B, r), where N is a set of nodes, $B \subset N \times N$ is a set of branches, and $r \in N$ is the root*

[1] We use the terms *vertex* and *edge* for a graph, and the terms *node* and *branch* for an index. Additionally, we use the term *CodeTree* to refer to the method for the graph search and the term *Code Tree* to refer to the index for the graph search.

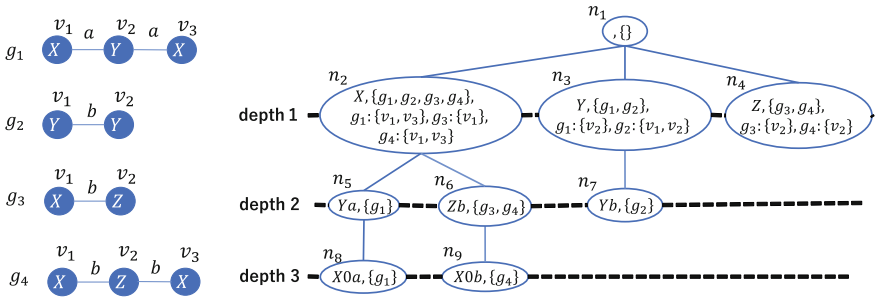


Fig. 2. Example of the Graph Database and Code Tree

of the tree. Each node in the Code Tree has a code fragment and set of graphs. If a code generated by concatenating the fragments associated with the nodes on the path from the root to node n is c and $g(c) \sqsubseteq g_i$ for $g_i \in D$, the set of graphs for n contains g_i. Additionally, each node with the vertex label ℓ at depth 1 has a set of vertices with the label ℓ in each g_i. ■

We denote the code fragment and set of graphs for node n by $fr(n)$ and $G_S(n)$, respectively. Additionally, we denote the graph represented by the code generated by concatenating fragments associated with nodes on the path from the root to node n by $g(n)$. From the Code Tree, we obtain a set of vertices in g_i at node n using $\Lambda(n, g_i)$. For example, for the database that consists of four graphs in Fig. 2(a), one possible code tree is depicted in Fig. 2(b). For n_2 in Fig. 2(b), $fr(n_2) = Y$ and $G_S(n_2) = \{g_1, g_2\}$. Additionally, $\Lambda(n_2, g_2) = \{v_1, v_2\}$.

The Code Tree stores a set of patterns for filtering. Constructing mining-based and enumeration-based indices for the subgraph search requires a huge computation time and significant memory space [18]. Additionally, when the database of graphs is updated, major reconstruction of the indices is required. To avoid these issues, we construct our Code Tree based on the following strategy.

1. We "randomly" generate a connected induced subgraph g_i^s with δ vertices from each $g_i \in G$.
2. We "randomly" generate an AcGM code $c_i \in \Omega(g_i^s)$ of g_i^s.
3. We add c_i such that c_i forms a path from the root of the Code Tree to node n.
4. To filter out graphs with infrequent labels, for each $\ell \in \Sigma$, we create an AcGM code ℓ and apply process (3) to ℓ.

Because only one code is generated from each graph in the database, the time-consuming processes of subgraph mining and enumeration are not required. Despite the simplicity of our indexing method, our subgraph search technique achieves highly efficient filtering and is comparable with VEQ$_S$ in its search processing, as will be demonstrated in evaluation experiments.

Lemma 2. *The number of nodes in the Code Tree and its space complexity are bounded by $\mathcal{O}(|G|\delta)$ and $\mathcal{O}(|G|^2\delta + \sum_{g_i \in G} |V(g_i)|)$, respectively.* ■

Algorithm 3: constructCodeTree

Input : a set of graphs G and the length δ of codes
Output: a code tree $\mathcal{T}$

1 $\mathcal{T} \leftarrow (\{r\}, \emptyset, r)$;
2 **for** $g_i \in G$ **do**
3 $\quad s_1 s_2 \cdots s_\delta \leftarrow getCode(g_i, \delta)$;
4 $\quad addPathToTree(s_1 s_2 \cdots s_\delta, \mathcal{T})$;
5 **for** $\ell \in \Sigma$ **do**
6 $\quad addPathToTree(\ell, \mathcal{T})$;
7 $\mathcal{T} \leftarrow pruningEquivalentNodes(\mathcal{T})$;
8 **for** $g_i \in G$ **do**
9 $\quad inclusionCheck(g_i, r, \langle\rangle)$;
10 **return** $\mathcal{T}$;

Algorithm 4: inclusionCheck

Input : a graph g_i, node n, and $\langle w_1, \ldots, w_h \rangle$ to generate a code from g_i

1 $G_S(n) \leftarrow G_S(n) \cup \{g_i\}$;
2 **if** $depth(n) = 1$ **then**
3 $\quad V' \leftarrow \{v \mid v \in V(g_i), \ell(v) = fr(n)\}$;
4 $\quad$ add "$g_i : V'$" in the node n;
5 **if** $\nexists m$ *s.t.* m *is a descendant of* n *and* $g_i \notin G_S(m)$ **then**
6 $\quad$ **return**;
7 $C \leftarrow \{(w, s) \mid s_1 \cdots s_h s = code(g_i, \langle w_1, \ldots, w_h, w \rangle)$ is a prefix of $c, c \in \Omega(g_i)\}$;
8 $N \leftarrow children(n)$;
9 **for** $(m, (w, c)) \in N \times C$ **do**
10 $\quad$ **if** $compare(fr(m), c)$ **then**
11 $\quad\quad inclusionCheck(g_i, m, \langle w_1, \ldots, w_h, w \rangle)$;

The AcGM codes stored in the Code Tree share their prefixes; hence, the actual number of nodes in the tree is much less than $|G|\delta$. By contrast, the number of patterns generated by subgraph mining or enumeration increases exponentially with the number of vertices in graphs contained in the database. Therefore, the index of the proposed method is very compact.

Algorithm 3 contains the pseudocode used to construct our Code Tree $\mathcal{T}$. Line 1 defines the root of $\mathcal{T}$. On Line 3, a prefix of length δ (a sequence of δ code fragments) is generated from among the AcGM codes of each graph $g_i \in G$. On Line 4, the prefix is added to the Code Tree to form a path from the root to a node at depth δ. By repeating Lines 5–6 for $|G|$ AcGM codes of length δ and $|\Sigma|$ codes of length 1, the Code Tree (which is our index) is constructed. If there are two or more leaf nodes corresponding to a certain graph, Line 7 of *pruningEquivalentNodes* prunes as many nodes as possible, leaving at least one. At this moment, $G_S(n)$ is empty for each node n. Line 9 finds nodes n that satisfy $g(n) \sqsubseteq g_i$ for each graph $g_i \in G$ and adds g_i to $G_S(n)$. Algorithm 4 generates all

possible AcGM codes from g in a depth-first manner, starting from each vertex in g. Each of the possible AcGM codes c is limited to that for a connected induced subgraph (but not a connected subgraph) of g for which there exist nodes n that satisfy $g(n) \sqsubseteq g(c)$. Generating all possible AcGM codes is equivalent to taking a permutation of the vertices in g_i. However, if the tree is compact, the time required to generate the codes is not significant because the diversity of the codes is limited. The procedures on Lines 5–6 of Algorithm 4 prevent the redundant traversal of a tree that does not update $G_S(n)$. The function *compare* on Line 10 is the same as that in [9].

The characteristics of the Code Tree are as follows:

1. Patterns in the Code Tree are connected induced subgraphs generated at random from $g_i \in G$. The number of patterns is $|G|$.
2. The patterns are included in $g_i \in G$ as induced subgraphs, but not as subgraphs. Graphs included as induced subgraphs have no fewer edges than those included as subgraphs, which is effective for filtering according to Lemma 1.
3. Connected induced subgraphs generated at random from $g_i \in G$ are stored in the Code Tree. No process for selecting the patterns is required.
4. These multiple codes represent each pattern g_i^s, and one of them is selected at random.
5. The number of nodes in the Code Tree is bounded by $\mathcal{O}(|G|\delta)$.

6 Subgraph Search with the Code Tree

In this section, we describe a method for traversing the Code Tree to obtain patterns contained in query q and candidate solutions Can, which corresponds to the supergraph search problem. The problem is to output $\{p \in P \mid p \sqsubseteq q\}$ from query q and graphs $P = \{p_1, \ldots .p_m\}$, all of which are stored in the index as patterns. Therefore, the pseudocode shown in Algorithm 6 is based on the supergraph search method proposed in [9]. The first characteristics of our proposed method are not only filtering graphs in G but also filtering nodes in each graph in G to reduce the number of nodes in the graph before verification, which reduces the computation time for verification. We call this *node filtering*. Second, when the method visits node n that satisfies $q = g(n)$ while traversing the tree, the graphs in $G_S(n)$ are added to S because they are solutions. When the number of graphs in Can has been sufficiently reduced, the computation time for verification is greatly reduced, similar to Lindex+ [23].

Algorithms 5 and 6 contain the pseudocode for the subgraph search based on the above characteristics. Algorithm 5 is based on Algorithm 1. On Line 3, our method traverses the Code Tree to obtain a subset of solutions S and set of candidate solutions Can. On Line 2, M is initialized with nodes at depth 1 in the Code Tree. Immediately after Line 3, there are still nodes in M that were not visited in the traversal. Vertices with labels that the nodes have are the target of node filtering that is executed on Line 5. On Line 6, the subgraph isomorphism problem is solved for the node-filtered graph g_i' and query graph q. Lines 10–14 of Algorithm 6 generate code fragments s for connected and induced subgraphs

Algorithm 5: search

Input : query graph q and Code Tree $\mathcal{T} = (N, B, r)$
Output: a set of solution $S = \{g_i \in G \mid q \sqsubseteq g_i\}$

```
1 S ← ∅, Can ← G;
2 M ← children(r);
3 traverse(q, r, ⟨⟩, S, Can, M, true);
4 for g_i ∈ Can \ S do
5 |   g'_i ← (V(g_i) \ ⋃_{n∈M} Λ(n, g_i), E(g_i) \ (V(g_i) × ⋃_{n∈M} Λ(n, g_i)), ℓ);
6 |   if verification(g'_i, q)=true then
7 |   |   S ← S ∪ {g_i};
8 return S;
```

of q and traverse nodes m that satisfy $g(m) \sqsubseteq q$. While traversing the Code Tree, Line 3 filters out graphs that are not solutions. If Can becomes empty, our method backtracks. *mode* is true if and only if $g(n) \sqsubseteq_i q$ but not if $g(n) \sqsubseteq q$. When our method visits node n that satisfies $q = g(n)$ using the value of *mode*, Lines 1 and 2 add graphs in $G_S(n)$ to S. Lines 6 and 7 prune the search space without changing S and Can according to the following lemma.

Lemma 3. *At node n in the Code Tree, if $S \neq \emptyset$ and $(Can \setminus S) \cap G_S(n) = \emptyset$, S and Can are unchanged at the descendant nodes of n.* ■

7 Experimental Evaluation

7.1 Experimental Settings

We compared the performance of our CodeTree$_{sub}$ with that of GGSX [1], GRAPES [7], VEQ$_S$ [12], and CFQL [18]. We conducted experiments on a machine running an AMD Ryzen Treadripper 3970X 32-Core processor with 128 GB RAM. CFQL and VEQ$_S$ are index-free subgraph search methods that use filtering to construct auxiliary data structures, and then verify the subgraph isomorphism between queries and candidate solutions. GGSX and GRAPES are subgraph search methods based on IFV. They enumerate all paths of length $\delta' = 4$ from graphs in G and then construct indices. We obtained executable files for GGSX, GRAPES, VEQ$_S$, and CFQL that run on Linux. These were implemented in C++. VEQ$_S$ is the fastest existing subgraph search method.

We implemented our method in Java[2]. We used VEQ$_S$ in our verification, but its source code is not available. Thus, we did the following.

1. We obtained Can by filtering using our method.
2. We wrote the graphs in Can and the query graph to a file.
3. We measured the computation time required by VEQ$_S$ for the file.

[2] The executable files written in Java and the datasets for evaluation are available at https://github.com/KG-CodeTree/CodeTree.

Algorithm 6: traverse

Input : query graph q, node n, $\langle w_1, \ldots, w_h \rangle$ to induce a code from q, a set of solutions S, a set of candidates Can, a set of nodes M, and $mode$

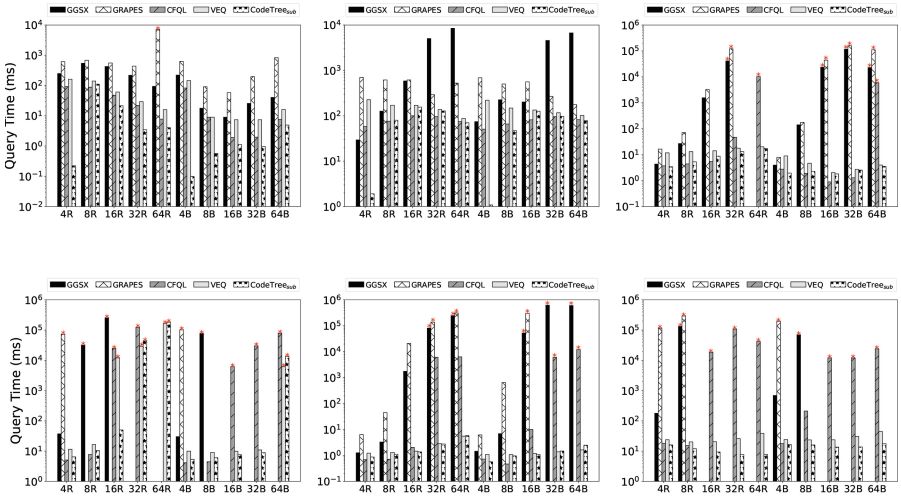
if $mode = true \wedge depth(n) = |V(q)|$ **then**
  $S \leftarrow S \cup G_S(n)$;
$Can \leftarrow Can \cap G_S(n)$;
if $Can = \emptyset \vee \nexists m$ *s.t.* m *is a descendant of* n *and has not yet been visited* **then**
  return;
if $S \neq \emptyset \wedge (Can \setminus S) \cap G_S(n) = \emptyset$ **then**
  return;
if $depth(n) = 1$ **then**
  $M \leftarrow M \setminus \{n\}$;
$C \leftarrow \{(w, s) \mid s_1 \cdots s_h s = code(q, \langle w_1, \ldots, w_h, w \rangle)$ is a prefix of $c, c \in \Omega(q)\}$;
$N \leftarrow children(n)$;
for $(m, (w, c)) \in N \times C$ **do**
  if $compare(fr(m), c)$ **then**
    $traverse(q, m, \langle w_1, \ldots, w_h, w \rangle, S, Can, M, mode \wedge (fr(m) = c))$;

The computation time of the proposed method is the time required for the above process, excluding the time required for file I/O. The computation time for verification in the proposed method includes the time required by VEQ_S to construct auxiliary data structures.

Query Sets: Each query graph was generated from $g \in G$ using either a breadth-first search (BFS) or random walk [18]. The specific procedure for generating the query graph is as follows: (1) select graph g at random from G, (2) select vertex v in g at random, (3) add every vertex and edge to the query graph generated by a BFS or random walk starting from v visits, and then (4) return the query when it has visited the predefined number of edges. Each query set $Q_{\epsilon B}$ (BFS) or $Q_{\epsilon R}$ (random walk) consisted of 100 graphs, where $\epsilon \in \{4, 8, 16, 32, 64\}$ represents the number of edges in each query. We call a query set in which the number of edges is small (large) a "small (large) query set". Because the subgraph search problem is NP-complete, we set a time limit of 10 min to process one query, similar to the experiments conducted by Kim et al. [12]. If the query could not be processed within the time limit, the query processing time (QPT) was recorded as 10 min.

Table 1. Benchmark Datasets

	$\lvert G \rvert$	$\lvert \Sigma \rvert$	$\overline{\lvert V(g) \rvert}$	$\overline{\lvert E(g) \rvert}$	*degree*	$\overline{\lvert \Sigma \rvert}$
AIDS	40,000	62	45	47	2.09	4.4
PDBS	600	10	2,939	3,064	2.06	6.4
PCM	200	21	377	4,340	23.01	18.9
PPI	20	46	4,942	26,667	10.87	28.5
IMDB	1500	10	13	66	10.14	6.9
REDDIT	4,999	10	509	595	2.34	10.0



(d) PPI (e) IMDB (f) REDDIT

Fig. 3. Query Processing Time on the Benchmark Datasets

Datasets: We used six benchmark datasets (AIDS, PDBM, PCM, PPI, IMDB, and REDDIT), as applied in previous studies [12,18]. AIDS consists of chemical compounds. PDBS consists of DNA, RNA, and protein structures. PCM and PPI consist of graphs that represent protein–protein interactions; the graphs in PPI are much larger than those in PCM. IMDB is a movie collaboration dataset. REDDIT is a dataset of online discussion communities. IMDB and REDDIT do not contain vertex labels. Thus, one of ten distinct labels was randomly assigned to each vertex [12]. In this paper, we do not provide results for the COLLAB dataset because most methods cannot return solutions for many of the queries within the time limit. Table 1 presents a summary of the datasets. $\overline{|V(g)|}$ and $\overline{|E(g)|}$ are the average numbers of vertices and edges of graphs in G, respectively. The *degree* is the average degree and $\overline{|\Sigma|}$ is the average number of vertex labels in a graph. VEQ$_S$ cannot treat edge labels; hence, we removed edge labels from the datasets.

7.2 Experimental Results

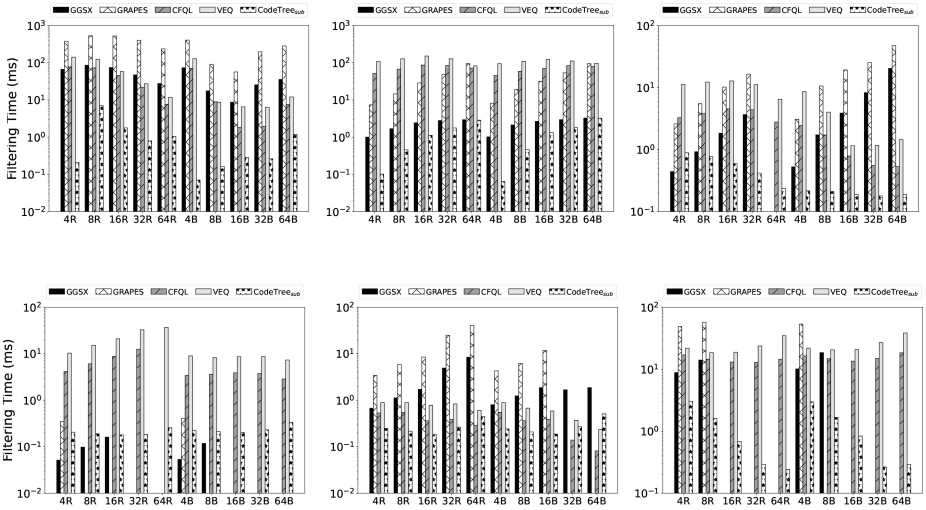
First, we present the results for the QPT, which reflects the core aim of this study. Then we examine detailed results related to query processing and experimental results related to index construction.

Figure 3 shows the QPTs for various query sets on each dataset. A red asterisk indicates that some queries did not yield results within the time limit. If the time limit was exceeded for more than 50 of the 100 queries in each query set, we provided no bar chart for that query set. For each of the six datasets, there were 10 query sets; that is, there were $6 \times 10 = 60$ test cases. In 53 of the 60 cases, the QPTs for CodeTree$_{sub}$ were shorter than those for VEQ$_S$. Note that we did not count the three cases in which both CodeTree$_{sub}$ and VEQ$_S$ were marked using a red asterisk. CodeTree$_{sub}$ performed well for small queries, and the two datasets AIDS and REDDIT. The reason that CodeTree$_{sub}$ performed well for small queries is that CodeTree$_{sub}$ found solutions while filtering. Table 2 presents the search precision results for various datasets and query sets. The search precision is the percentage of solutions that the proposed method found while filtering and is defined as $\frac{1}{|Q|}\sum_{q \in Q} \frac{|In(q)|}{|S(q)|}$, where $|In(q)|$ is the number of solutions found by Algorithm 6, but not by Algorithm 5. There are no IFV methods other than CodeTree$_{sub}$ for which search precision is greater than 0. The different values of δ correspond to tuning the QPT of CodeTree$_{sub}$ to be shorter. When the smallest number of vertices in the graphs in query set Q is larger than the depth δ of the Code Tree, the search precision is always zero. We did not include such cases in Table 2. For Q_{4R} and Q_{4B} in the AIDS and PDBS datasets, Algorithm 6 returned many solutions. This is because there were many nodes n up to a depth of 5 in the Code Tree and the graphs $g(n)$ stored in the tree were very diverse. When many solutions were found while filtering, the number of graphs to be verified was greatly reduced and the computation time for verification reduced accordingly. By contrast, the reason that CodeTree$_{sub}$ performed well for the AIDS and REDDIT datasets is that the densities of graphs in the datasets were small. CodeTree$_{sub}$ selected induced subgraph of graphs in G as patterns to be registered in the index. Because the induced subgraph had many edges, the induced subgraph filtered out sparse graphs in G for a given query graph. In [12], it has already been mentioned that CFQL and VEQ$_S$, which are methods based on vcFV, outperformed GGSX and GRAPES, which are methods based on IFV. For this reason, indices were not used in [18]. However, CodeTree$_{sub}$ is an IFV-based method, and CodeTree$_{sub}$ is as fast as or faster than VEQ$_S$ or CFQL. Therefore, IFV-based subgraph methods are not necessarily slower, and in this paper, we showed that existing methods have room for improvement.

Table 2. Search Precision

Dataset	Query	δ	Search Precision
AIDS	Q_{4R}	5	0.97
AIDS	Q_{4B}	5	0.99
PDBS	Q_{4R}	20	0.59
PDBS	Q_{8R}	20	0.16
PDBS	Q_{16R}	20	0.01
PDBS	Q_{4B}	20	0.72
PDBS	Q_{8B}	20	0.35
PDBS	Q_{16B}	20	0.01
PCM	Q_{8R}	10	0.03

Figure 4 shows the filtering times for the various datasets. The filtering times of CodeTree$_{sub}$ depend on the sizes of the graphs in the query sets and the number of nodes in the Code Tree. When there were few nodes in the Code Tree, the filtering time of CodeTree$_{sub}$ was small because the search space for the queries became narrower, although CodeTree$_{sub}$ filtered out relatively few



(d) PPI (e) IMDB (f) REDDIT

Fig. 4. Filtering Time on the Benchmark Datasets

Table 3. Constructing Indices using IFV-based Methods

	δ	# of nodes in the Code Tree	Time for constructing index [s]			Index size [MB]		
			CodeTree$_{sub}$	GGSX	GRAPES	CodeTree$_{sub}$	GGSX	GRAPES
AIDS	5	2,558	18.0	26	12	17.1	28	39
PDBS	20	5,711	65.8	19	4	2.6	20	17
PCM	10	1,555	3.2	1340	233	0.6	312	1360
PPI	5	115	0.1	6194	936	0.4	23	664
IMDB	4	1553	0.3	57	11	0.9	29	38
REDDIT	3	674	16.3	1645	280	5.0	232	1580

graphs. As the size of the query graph increased, the filtering time of CodeTree$_{sub}$ became large because the search space for the queries increased as the number of codes generated from the queries increased. By contrast, when the size of the query graph increased, the filtering times of CFQL and VEQ$_S$ decreased. This is because the construction of auxiliary structures involves filtering based on vertex connectivity. Effectively, the vertices in the query have more adjacent vertices, and non-solution graphs can be filtered earlier.

Table 3 presents several details about the construction of indices for IFV-based methods. The different values of δ are the result of tuning the QPT of CodeTree$_{sub}$ to be shorter. The number of nodes in the Code Tree depends on δ, $|G|$, and the characteristics of the dataset. Compared with GGSX and GRAPES, CodeTree$_{sub}$ takes less time to construct indices and requires less memory to hold

the indices. GGSX and GRAPES enumerate all paths of a certain length from the graph in G and store them in indices, which makes the size of their indices larger and their construction time longer.

8 Conclusion

We proposed an index-based subgraph search method that is as fast as the current state-of-the-art technique. CodeTree$_{sub}$ treats graphs as graph codes and uses the prefix tree for these graph codes as an index. This index permits the highly efficient filtering of non-solutions, but its construction entails little computational overhead. CodeTree$_{sub}$ effectively limits the number of candidate solutions so that only induced subgraphs of graphs in databases are included in the index, thus accelerating the filtering step. Additionally, CodeTree$_{sub}$ can identify some solutions during the filtering stage. We compared the performance of CodeTree$_{sub}$ on six benchmark datasets. The results demonstrated that the proposed method was consistently as fast as or faster than the state-of-the-art VEQ$_S$ method in terms of query processing. This study is of particular interest because it illustrates that index-based methods have the potential to outperform non-index-based techniques, thereby providing enhanced query speeds for small- and large-scale databases alike.

References

1. Bonnici, V., et al.: Enhancing graph database indexing by suffix tree structure. In: Proceedings of International Conference on Pattern Recognition in Bioinformatics, pp. 195–203 (2010)
2. Chen, C., et al.: Towards graph containment search and indexing. In: Proceedings of International Conference on Very Large Data Bases, pp. 926–937 (2007)
3. Cheng, J., et al.: FG-index: towards verification-free query processing on graph databases. In: Proceedings of International Conference on Management of Data, pp. 857–872 (2007)
4. Cheng, J., et al.: Efficient query processing on graph databases. ACM Trans. Database Syst. **34**(2), 1–48 (2009)
5. Cordella, L., et al.: A (sub)graph isomorphism algorithm for matching large graphs. IEEE Trans. Pattern Anal. Mach. Intell. **26**(10), 1367–1372 (2004)
6. Garey, M., Johnson, D.: Computers and Intractability: A Guide to the Theory of NP-Completeness. H. Freeman (1979)
7. Giugno, R., et al.: GRAPES: a software for parallel searching on biological graphs targeting multi-core architectures. PloS One **8**(10) (2013)
8. He, H., Singh, A.: Graphs-at-a-time: query language and access methods for graph databases. In: Proceedings of International Conference on Management of Data, pp. 405–418 (2008)
9. Imai, S., Inokuchi, A.: Efficient supergraph search using graph coding. IEICE Trans. Inf. Syst. **103–D**(1), 130–141 (2020)
10. Inokuchi, A., et al.: A Fast algorithm for mining frequent connected subgraphs. IBM Research Report, RT0448, IBM Research (2002)

11. Kim, H., et al.: IDAR: fast supergraph search using DAG integration. Proc. of VLDB Endow. **13**, 1456–1468 (2020)
12. Kim, H., et al.: Versatile equivalences: speeding up subgraph query processing and subgraph matching. In: Proceedings of International Conference on Management of Data, pp. 925–937 (2021)
13. Klein, K., et al.: CT-index: fingerprint-based graph indexing combining cycles and trees. In: Proceedings of International Conference on Data Engineering, pp. 1115–1126 (2011)
14. Lyu, B., et al.: Scalable supergraph search in large graph databases. In: Proceedings of International Conference on Data Engineering, pp. 157–168 (2016)
15. Natale, R., et al.: SING: subgraph search in non-homogeneous graphs. BMC Bioinform. **11**(96) (2010)
16. Qin, Z., et al.: GHashing: semantic graph hashing for approximate similarity search in graph databases. In: Proceedings of ACM SIGKDD Conference on Knowledge Discovery and Data Mining, pp. 2062–2072 (2020)
17. Shasha, D., et al.: Algorithmics and applications of tree and graph searching. In: Proceedings of Symposium on Principles of Database Systems, pp. 39–52 (2002)
18. Sun, S., Luo, Q.: Scaling up subgraph query processing with efficient subgraph matching. In: Proceedings of IEEE International Conference on Data Engineering, pp. 220–231 (2019)
19. Williams, D., et al.: Graph database indexing using structured graph decomposition. Proceedings of IEEE International Conference on Data Engineering, pp. 976–985 (2007)
20. Xie, Y., Yu, P.: CP-index: on the efficient indexing of large graphs. In: Proceedings of ACM Conference on Information and Knowledge Management, pp. 1795–1804 (2011)
21. Yan, X., Han, J.: gSpan: graph-based substructure pattern mining. In: Proceedings of IEEE International Conference on Data Mining (ICDM), pp. 721–724 (2002)
22. Yan, X., et al.: Graph indexing: a frequent structure-based approach. In: Proceedings of ACM SIGMOD International Conference on Management of Data, pp. 335–346 (2002)
23. Yuan, D., Mitra, P.: Lindex: a lattice-based index for graph databases. VLDB J. 229–252 (2013)
24. Zhao, P., et al.: Graph indexing: tree + delta >= graph. In: Proceedings of International Conference on Very Large Data Bases, pp. 938–949 (2007)
25. Zhu, Y., et al.: Answering Top-k k graph similarity queries in graph databases. IEEE Trans. Knowl. Data Eng. **32**(8), 1459–1474 (2020)