IEICE TRANS. INF. & SYST., VOL.E103–D, NO.1 JANUARY 2020

PAPER

Efficient Supergraph Search Using Graph Coding

Shun IMAI[†], *Nonmember and* **Akihiro INOKUCHI**[†a)], *Member*

SUMMARY This paper proposes a method for searching for graphs in the database which are contained as subgraphs by a given query. In the proposed method, the search index does not require any knowledge of the query set or the frequent subgraph patterns. In conventional techniques, enumerating and selecting frequent subgraph patterns is computationally expensive, and the distribution of the query set must be known in advance. Subsequent changes to the query set require the frequent patterns to be selected again and the index to be reconstructed. The proposed method overcomes these difficulties through graph coding, using a tree structured index that contains infrequent subgraph patterns in the shallow part of the tree. By traversing this code tree, we are able to rapidly determine whether multiple graphs in the database contain subgraphs that match the query, producing a powerful pruning or filtering effect. Furthermore, the filtering and verification steps of the graph search can be conducted concurrently, rather than requiring separate algorithms. As the proposed method does not require the frequent subgraph patterns and the query set, it is significantly faster than previous techniques; this independence from the query set also means that there is no need to reconstruct the search index when the query set changes. A series of experiments using a real-world dataset demonstrate the efficiency of the proposed method, achieving a search speed several orders of magnitude faster than the previous best.

key words: *supergraph search, indexing, graph coding, canonical form, subgraph isomorphism*

1. Introduction

Graph searches are a fundamental component in applications such as chemo-informatics [19], [26], bio-informatics [1], computer-aided design [14], computer vision [16], [18], pattern recognition, XML, social networks, World Wide Web, and software analysis. For a database consisting of the set of graphs $G = \{g_1, g_2, \cdots, g_n\}$ and a given query q, there are two types of graph searches depending on the desired output:

- Subgraph search: $\{g_i \in G \mid q \subseteq g_i\}$ [3], [7], [8], [11], [17], [22], [24]
- Supergraph search: $\{g_i \in G \mid g_i \subseteq q\}$ [4], [13], [21], [23], [25].

where $p \subseteq g$ indicates that p is a subgraph of g. In this paper, we focus on supergraph searches.

A labeled graph is represented by $g = (V, E, L, \ell)$, where V is a set of vertices, $E \subseteq V \times V$ is a set of edges, L is a set of labels, and $\ell : V \cup E \to L$ is a function that assigns a label to each vertex or edge in the graph. Additionally, the sets of vertices and edges in graph g are represented by $V(g)$ and $E(g)$, respectively. We denote the degree of a vertex v in a graph g as $d(g, v)$.

Definition 1 (Subgraph): Given two graphs $g = (V, E, L, \ell)$ and $g' = (V', E', L', \ell')$, g' is called a subgraph of g if there exists an injective function (one-to-one mapping) $\phi : V' \to V$ that satisfies the following three conditions $\forall v, v_1, v_2 \in V'$:

(1) $\ell'(v) = \ell(\phi(v))$.
(2-1) $(\phi(v_1), \phi(v_2)) \in E$, if $(v_1, v_2) \in E'$.
(2-2) $\ell'((v_1, v_2)) = \ell((\phi(v_1), \phi(v_2)))$.

The subgraph isomorphism problem is the NP-complete problem of finding mappings between g and g'. The set Φ of all possible permutations of integers $1, 2, \cdots, |V|$ may contain many mappings that satisfy the conditions. g is called a supergraph of g' when $g' \subseteq g$. Additionally, a subgraph g' of g is an *induced* subgraph if and only if two vertices in $V(g')$ are adjacent in g' and are also adjacent in g. When a graph is isomorphic to itself, this is referred to as an automorphism. In this paper, we deal with connected undirected graphs, although the proposed method can be extended to unconnected graphs or directed graphs.

Given a graph p, the support $\sigma(p, G)$ for p is defined as

$$\sigma(p, G) = |\{i \mid g_i \in G, p \subseteq g_i\}|.$$

Furthermore, given a certain threshold σ', the graph p that satisfies $\sigma(p, G) \geq \sigma'$ is referred to as a frequent subgraph pattern [9]. In this paper, the set of all frequent subgraph patterns enumerated from G is written as $F = \{p \mid p \subseteq g_i, g_i \in G, \sigma(p, G) \geq \sigma'\}$.

A supergraph search attempts to identify graphs in G satisfying $g_i \subseteq q$ for a given query q. As the subgraph isomorphism problem is NP-complete [5], [6], solving it for each graph in G and q is very inefficient. Therefore, in conventional method, the following properties are often used:

- For a graph $p \not\subseteq q$, g_i cannot be a subgraph of q if $p \subseteq g_i$.
- When the number of vertices in p is smaller than the number of vertices in $g_i \in G$, it is quicker to determine whether $p \subseteq q$ than whether $g_i \subseteq q$.

Therefore, for $P = \{p \mid p \not\subseteq q\} \subseteq F$, if we first seek

$$G' = G - \bigcup_{p \in P} \{g \mid g \in G, p \subseteq g\}$$

Manuscript received January 10, 2019.
Manuscript revised July 20, 2019.
Manuscript publicized September 26, 2019.

[†]The authors are with Department of Science and Engineering, Kwansei Gakuin University, Sanda-shi, 669–1337 Japan.
a) E-mail: inokuchi@kwansei.ac.jp

DOI: 10.1587/transinf.2019EDP7011

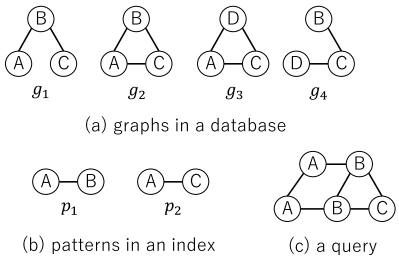

Fig. 1 Example of supergraph search.

and then solve the subgraph isomorphism problem for each graph of G' and q, the search can be accelerated. The process in which $\cup_{p \in P}\{g \mid g \in G, p \subseteq g\}$ is removed from G is known as *filtering*. Furthermore, solving the subgraph isomorphism problem for each graph in G' and q is known as *verification*. Various methods have been developed for choosing patterns from F to filter as many graphs in G as possible. Representative methods for this problem are the LW-index [21], PrefIndex [25], GPTree [23], and IG-Query [4].

Figure 1 (a) shows four graphs in a database, and two patterns chosen from F (see Fig. 1 (b)) are contained in an index. The index has information that $p_1 \subset g_1$, $p_1 \subset g_2$, $p_2 \subset g_2$, and $p_2 \subset g_3$. Given the query shown in Fig. 1 (c), conventional supergraph search techniques first check whether $p_1 \subseteq q$ and $p_2 \subseteq q$, and then obtain $G' = \{g_1, g_4\}$ in the filtering phase. In the verification phase, they solve whether $g_1 \subseteq q$ and $g_4 \subseteq q$.

2. Issues with Conventional Methods

In this section, we identify the drawbacks of conventional methods and the characteristics of the proposed method.

- When using the LW-index, choosing an optimal subset of frequent subgraph patterns from F to be used in an index requires $O(k|F||Q||G|)$ operations, where k is the number of patterns chosen from F and Q is a set of queries. An improved method can select a suboptimal solution in $O(k|F||G|)$. However, the computation time depends on $|F|$. In addition, algorithms for enumerating frequent subgraph patterns are extremely time-consuming.
- For a query q, some frequent subgraph patterns p are considered to be suitable for filtering graphs that are not included in q. However, when p appears frequently in G, there is a high probability that p will also appear frequently in Q. Thus, it is not necessarily true that frequent subgraphs are suitable for filtering as mentioned in the article [27]. In fact, some infrequent subgraphs may be better suited to filtering.
- Under the assumption that the distribution of Q is known in advance of constructing an index, many conventional methods choose some frequent subgraph patterns that are considered suitable for indexing. For this reason, they cannot be applied when Q is not known in advance. Moreover, when the distribution of Q changes, it is necessary to choose the patterns again and construct another index.
- There may be many mappings between a graph g_i in the database and a pattern p that is a subgraph of g_i. As the index stores all of the mappings between g_i and p, the index size can become very huge.

To overcome these drawbacks, we propose a method that does not require F and Q when constructing an index, but achieves a few orders of magnitude faster than the LW-index. The characteristics of the proposed method are as follows.

- F and Q are not required, and the computation time required to construct the index is $O(|G|)$.
- The proposed index has a tree structure, and there are infrequent subgraph patterns in the shallow part of the tree, which is similar to feature-based index [17].
- As Q is not required when constructing the index, there is no need to reconstruct the index if the distribution of Q changes. Furthermore, even if some graphs are added to, deleted from, or updated in G, the index can be changed simply.
- As it does not store any mappings between graphs in the databases and patterns in the index, our index is very compact.
- The filtering and verification are computed in the same algorithm rather than through individual, specific algorithms.
- Searching is a few orders of magnitude faster than the current fastest LW-index.

The remainder of this paper is organized as follows. In Sect. 3.1, we define some graphs codes to represent labeled graphs. In Sect. 3.2, we propose an index called a code tree to store graphs in a database and discuss the insertion, deletion, and update of database elements. In Sect. 3.3, we propose a method of searching for graphs that are subgraphs of the given query by traversing the code tree. In Sect. 3.4, we explain some methods for optimizing our proposed graph search method. In Sect. 4, we verify the computational efficiency of the proposed method and compare it with the conventional method in terms of computation time using a real-world dataset. Finally, we conclude this paper in Sect. 5.

3. Proposed Method

3.1 Graph Representation

We define three types of graph code. These codes are used in frequent subgraph mining algorithms such as AcGM [10] and gSpan [20].

Given a graph $g = (V, E, L, \ell)$, the vertices in V are assigned ID numbers from 1 to $|V|$ and denoted as $v_1, v_2, \cdots, v_{|V|}$. g is represented by an adjacency matrix. If $(v_i, v_j) \in E$, then the (i, j)-th element of the matrix is $\ell((v_i, v_j))$; in all other cases, this is zero.

Definition 2 (AcGM code): IDs from 1 to $|V|$ are assigned to vertices in V such that a subgraph induced by vertices $v_1, v_2, \cdots, v_i$ $(1 \leq i \leq |V|)$ must be connected. By laying out elements in the upper triangular portion of the adjacency matrix for the graph g, as shown in Fig. 2, the AcGM code of the graph g is defined as

$$code(g, \langle v_1, v_2, \cdots, v_{|V|} \rangle) = c_1 c_2 \cdots c_{|V|}, \quad (1)$$

where

$$c_i = \ell(v_i) x_{1,i} x_{2,i} \cdots x_{i-2,i} x_{i-1,i}.$$

We call c_i a code fragment, and say that $c_1 c_2 \ldots c_i$ is a prefix of Eq. (1). Furthermore, when a prefix of code α is β, we write $\beta \subseteq \alpha$. If a prefix of the code of graph g is the same as the code of graph g', g' is an induced subgraph of g. There are several codes representing an isomorphic graph g. We denote a set of codes for g as $\Omega(g)$.

Definition 3 (AcGM code order): Given the AcGM codes $\alpha = a_1 a_2 \cdots a_k$ and $\beta = b_1 b_2 \cdots b_h$, $\beta \prec \alpha$ if and only if either of the following statements is true.

- There is some t that satisfies $1 \leq t \leq min(k, h)$, and $a_q = b_q$ and $b_t \prec_e a_t$ in relation to $q < t$.
- $\beta \subseteq \alpha$.

Here, $\prec_e$ represents the lexicographic order between the code fragments.

Example 1: The graphs shown in Fig. 3 (b), (c), and (d) have different vertex ID assignments for the vertices shown in Fig. 3 (a). The codes for the graphs are as shown below.

$$\alpha = X\ Xa\ Xab\ X0bc\ X0d00$$
$$\beta = X\ Xd\ X0b\ X0aa\ X0dc0$$
$$\gamma = X\ Xa\ Xba\ X00d\ Xc0b0$$

According to the definition of the code order, $\alpha \prec \gamma \prec \beta$.

$$\begin{pmatrix} \ell(v_1) & \ell(v_2) & \ell(v_3) & \cdots & \ell(v_{|V|}) \\ 0 & x_{1,2} & x_{1,3} & \cdots & x_{1,|V|} \\ x_{2,1} & 0 & x_{2,3} & \cdots & x_{2,|V|} \\ x_{3,1} & x_{3,2} & 0 & \cdots & x_{3,|V|} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{|V|,1} & x_{|V|,2} & x_{|V|,3} & \cdots & 0 \end{pmatrix}$$

Fig. 2 Adjacency matrix of a graph.

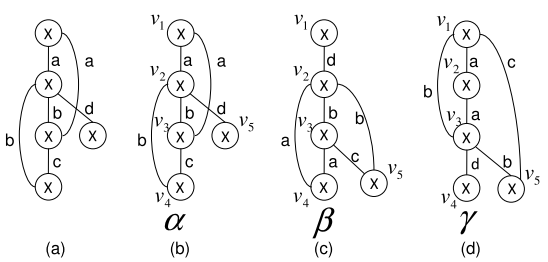

Fig. 3 Isomorphic graphs with different ways of assigning the vertex ID.

Lemma 1: If the AcGM codes $\alpha = a_1 a_2 \cdots a_k$ and $\beta = b_1 b_2 \cdots b_h$ are such that $a_i = \ell_i x_{1,i} \cdots x_{i-1,i}$ and $b_i = \ell'_i x'_{1,i} \cdots x'_{i-1,i}$ satisfy the following conditions for all $i \leq k$, the graph represented by α is a subgraph of the graph represented by β.

(1) $\ell_i = \ell'_i$.
(2) $x_{j,i} = x'_{j,i}$ or $x_{j,i} = 0$ for all $j < i$.

Proof 1: Let g and g' be the graphs expressed by α and β, respectively. $\ell_i = \ell'_i$ suggests that the label of vertex v_i in g is the same as that of vertex v'_i in g', which satisfies Definition 1(1) for a function ϕ where $\phi(i) = i$. In addition, $x_{j,i} = x'_{j,i}$ or $x_{j,i} = 0$ suggests that there is an edge between v'_i and v'_j when there is an edge between v_i and v_j, which satisfies Definition 1(2-1) for the function ϕ. Furthermore, $x_{j,i} = x'_{j,i}$ also suggests that the label of edge (i, j) in g is the same as the label of edge (i, j) in g', which satisfies Definition 1(2-2). Therefore, Lemma 1 has been proved because the injection where $\phi(i) = i$ exists between g and g'.

In this paper, we extend the AcGM code to a code with vertex degrees for the following reason. For $(v_1, v_2) \in V(g_1) \times V(g_2)$, if $d(g_1, v_1) > d(g_2, v_2)$, an injective function ϕ that maps v_1 to v_2 is not a mapping for $g_1 \subseteq g_2$. Thus, the degrees of the vertices provide important information for checking whether $g_1 \subseteq g_2$.

Definition 4 (extended AcGM code): IDs from 1 to $|V|$ are assigned to vertices according to Definition 2. We denote the connected subgraph induced by vertices $v_1, v_2, \cdots, v_i$ as g^i. By laying out elements in the upper triangular portion of the adjacency matrix for the graph g, the extended AcGM code (exAcGM code) of the graph g is defined as

$$code(g, \langle v_1, v_2, \cdots, v_{|V|} \rangle) = c_1 c_2 \cdots c_{|V|},$$

where

$$c_i = \ell(v_i) d(g^i, v_i) d(g, v_i) x_{1,i} x_{2,i} \cdots x_{i-2,i} x_{i-1,i}.$$

For example, the exAcGM code for the graph in Fig. 3 (b) is given as $X02\ X14a\ X23ab\ X220bc\ X110d00$.

Lemma 2: If the exAcGM codes $\alpha = a_1 a_2 \cdots a_k$ and $\beta = b_1 b_2 \cdots b_h$ are such that $a_i = \ell_i d_{i1} d_{i2} x_{1,i} \cdots x_{i-1,i}$ and $b_i = \ell'_i d'_{i1} d'_{i2} x'_{1,i} \cdots x'_{i-1,i}$ satisfy the following conditions for all $i \leq k$, the graph represented by α is a subgraph of the graph represented by β.

(1) $\ell_i = \ell'_i$.
(2) $x_{j,i} = x'_{j,i}$ or $x_{j,i} = 0$ for all $j < i$.
(3) $d_{i1} \leq d'_{i1}$ and $d_{i2} \leq d'_{i2}$.

Proof 2: This can be proved in the same way as Lemma 1.

We now define the depth-first search (DFS) code used in gSpan [20].

Definition 5 (DFS code): Given a graph g, a depth-first search of g is performed from a certain vertex in g. The

depth-first discovery of the vertices and edges forms lexicographic orders. We use subscripts to label these orders according to their discovery times. When $i < j$, v_i was discovered before v_j. Similarly, $a < b$ means that $e_a = (v, u)$ was discovered before $e_b = (v', u')$. The DFS code for graph g is defined as

$$code(g, \langle e_1, e_2, \cdots, e_{|E|} \rangle) = c_1 c_2 \cdots c_{|E|},$$

where c_i has information on edge (v, u) with two vertices v and u, and is defined as the 5-tuple $c_i = (v, u, \ell(v), \ell(u), \ell((v, u)))$.

The prefix of the DFS code, the relationship $\subseteq$, and the total-order relationship among the codes are defined in the same way as for the AcGM code. It's detailed definition of the DFS lexicographic order is given in [20].

Example 2: The depth-first search assigns vertex IDs to vertices in the graphs as shown in Fig. 3 (b), (c), and (d). The DFS codes for these graphs are presented in Table 1. According to the definition of the code order, $\gamma \prec \alpha \prec \beta$.

Lemma 3: If a DFS code α is a prefix of another DFS code β generated by traversing subsets of $V(g)$ and $E(g)$ in a graph g, the graph represented by α is a subgraph of the graph represented by β.

Proof 3: Let the DFS codes α and β be $\alpha = c_1 c_2 \cdots c_k$ and $\beta = c'_1 c'_2 \cdots c'_h$, where

$$c_i = (v_i, u_i, \ell(v_i), \ell(u_i), \ell((v_i, u_i))) \text{ and}$$
$$c'_i = (v'_i, u'_i, \ell'(v'_i), \ell'(u'_i), \ell'((v'_i, u'_i))).$$

If α is a prefix of β, $c_i = c'_i$ holds for all $i \leq k$. Therefore, $\ell(v_i) = \ell'(v_i)$, $\ell(u_i) = \ell'(u_i)$, and $\ell((v_i, u_i)) = \ell'((v_i, u_i))$, which satisfies Definitions 1(1), (2-1), and (2-2) for a function ϕ where $\phi(i) = i$. Therefore, Lemma 3 has been proved because the injection where $\phi(i) = i$ exists between g and g'.

When $g \in G$ is a subgraph of a query q, there must be at least one code for q that satisfies one of the above lemmas with respect to the code for g. Conversely, if there are no codes of q that satisfy the lemmas with the code for g, q is not a subgraph of g. Therefore, given a query q, we can consider a method for generating various codes of q and searching for graphs in G that satisfy one of the lemmas with respect to the code of q. However, as multiple codes are generated for q for each graph in the database, this straightforward method is very inefficient. Therefore, in the next subsection, we propose a code tree to share the subgraph isomorphic calculation between prefixes of multiple graphs in G and q and reduce the total computation time.

Table 1 DFS codes for graphs in Fig. 3.

c_i	α (Fig. 3 (b))	β (Fig. 3 (c))	γ (Fig. 3 (d))
c_1	$(1, 2, X, X, a)$	$(1, 2, X, X, d)$	$(1, 2, X, X, a)$
c_2	$(2, 3, X, X, b)$	$(2, 3, X, X, b)$	$(2, 3, X, X, a)$
c_3	$(3, 1, X, X, a)$	$(3, 4, X, X, a)$	$(3, 1, X, X, b)$
c_4	$(3, 4, X, X, c)$	$(4, 2, X, X, a)$	$(3, 4, X, X, d)$
c_5	$(4, 2, X, X, b)$	$(3, 5, X, X, c)$	$(3, 5, X, X, b)$
c_6	$(2, 5, X, X, d)$	$(5, 2, X, X, b)$	$(5, 1, X, X, c)$

3.2 Indexing the Graph Database

Definition 6 (code tree): We define the code tree T as $(\top, N, B)$, where $\top \in N$ is the root node, N is a set of nodes, and B is a set of branches. In addition, each node has a code fragment and a set of graph IDs, and we denote a concatenation of code fragments on the path from the root to the node n as $s(n)$. When a code for $g_i \in G$ is identical to $s(n)$, i is included in the set of IDs for the node n.

We denote the code fragments and ID sets of node n as $fr(n)$ and $ID(n)$, respectively. We assume that $fr(\top) = null$ and $ID(\top) = \emptyset$. An example of a code tree is presented below.

Example 3: The database $G = \{g_1, g_2, g_3, g_4\}$ consists of the four graphs shown in Fig. 4. We can generate various codes from one of the graphs, but when one of the codes is generated from each graph in G, the respective AcGM codes are

$$code(g_1, \langle v_1, v_2, v_3, v_4 \rangle) = X\ Xa\ X0a\ Xaaa,$$
$$code(g_2, \langle v_1, v_2, v_3, v_4 \rangle) = X\ Xa\ Xaa\ Xa00,$$
$$code(g_3, \langle v_1, v_2, v_3 \rangle) = X\ Xa\ X0a, \text{ and}$$
$$code(g_4, \langle v_1, v_2, v_3, v_4 \rangle) = X\ Xa\ Xa0\ X0aa.$$

The code tree T_1 for these graphs is shown on the right side of Fig. 4. Node n_5 has the code fragment $Xaaa$ and the graph ID set $ID(n) = \{1\}$. If we concatenate the code fragments on each node from the root node $\top = n_1$ to node n_5, we obtain $X\ Xa\ X0a\ Xaaa$, and the graph represented by the concatenation is g_1. Not only the code for g_1, all of the above four codes are represented by certain paths from the root node to other nodes whose IDs are not empty in T_1. The number of graph IDs included in all nodes of the code tree is equal to the number of graphs in the database.

Algorithm 1 shows the pseudocode for constructing a code tree for a given database G. Algorithm 1 is applicable to either AcGM, exAcGM, or DFS code without requiring any changes. Line 1 of Algorithm 1 initializes the code tree consisting of only root node. In Line 3, an arbitrary code

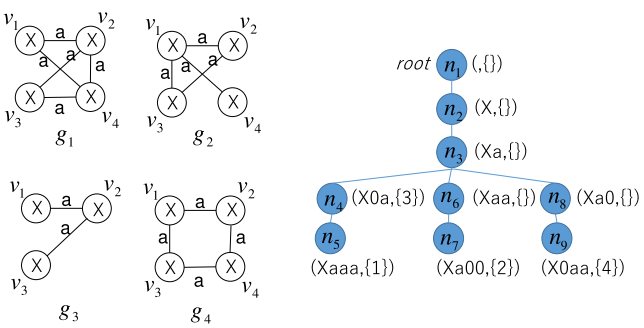

Fig. 4 Code tree example.

Algorithm 1: index

```
Data: a set of graphs G
Result: an index tree T
1 T ← (⊤, {⊤}, ∅);
2 for g_i ∈ G do
3     c_1 c_2 ··· c_k ← getCode(g_i);
4     addPathToTree(c_1 c_2 ··· c_k, ⊤, T, i);
5 return T;
```

Algorithm 2: addPathToTree

```
Data: code c_1 c_2 ··· c_h, an index tree T, the current node n, the
      graph ID id
1 if c_1 c_2 ··· c_h = null then
2     n ← (fr(n), ID(n) ∪ {id});
3     return ;
4 for m ∈ children(n) do
5     if fr(m) = c_1 then
6         addPathToTree(c_2 c_3 ··· c_h, m, T, id);
7         return ;
8 m ← createNode(c_1);
9 T ← (⊤, V(T) ∪ {m}, E(T) ∪ {(n, m)});
10 addPathToTree(c_2 ··· c_h, m, T, id);
```

is generated for each graph in G. The code is added to the tree such that the code becomes $s(n)$ for a certain node n using Algorithm 2. On Lines 1–3 of Algorithm 2, if the code is blank, the ID of g_i is added to the ID list for the current node n. Otherwise, the fragment for each child node of n is compared to c_1; if they match, we move to this child node in Line 6. Conversely, if there is no node whose fragment matches c_1, a new node is created as a child of n with fragment c_1 in Line 8 and added to the code tree in Line 9. To construct the code tree, Algorithm 1 does not require the set of all frequent subgraph patterns F enumerated from G or the set of queries Q. This is one of the characteristics of the proposed method, and it is different from conventional methods such as PrefIndex and LW-index. Therefore, it is not necessary to reconstruct the index, even if the distribution of Q changes,

Conventional supergraph search methods store all possible mappings between graphs in the database and patterns in their indices. For example, a pattern represented by node n_3 corresponds to four pairs of vertices (v_1, v_2), (v_2, v_1), (v_2, v_3), and (v_3, v_2) in g_3, ten pairs for g_1, eight pairs for g_2, and eight pairs for g_4. Using conventional methods, all of their mappings would be stored in the indices. In addition, because the patterns in the indices are frequent subgraph patterns in G, the memory space of the indices increases exponentially with the number of vertices in the graphs of the database. Conversely, the code tree does not store any mappings between graphs in the database and patterns represented by concatenated fragments.

Lemma 4: When graphs in the database are expressed in the AcGM or exACGM code, the memory space and number of nodes in the code tree are at most $O(\Sigma_{i=1}^{n}|V(g_i)|^2)$ and $O(\Sigma_{i=1}^{n}|V(g_i)|)$, respectively.

Proof 4: The memory space required to express a graph in the AcGM or exAcGM code is proportional to the size of the adjacency matrix for that graph. In addition, the memory space required by the adjacency matrix is proportional to the square of the number of vertices in the graph. Therefore, for the database D including n graphs, the memory space is at most $O(\Sigma_{i=1}^{n}|V(g_i)|^2)$. When adding a new graph g to a code tree, if there are no nodes n for which $s(n)$ is identical to the code of the graph, the number of nodes in the code tree increases by $|V(g)|$, because the number of nodes added is the same as the number of vertices in g. Therefore, the number of nodes in the code tree for the database D including n graphs is at most $O(\Sigma_{i=1}^{n}|V(g_i)|)$. Therefore, Lemma 4 has been proved.

Lemma 5: When graphs in the database are expressed in the DFS code, the size and number of nodes in the code tree are at most $O(\Sigma_{i=1}^{n}|E(g_i)|)$.

Proof 5: A graph with m edges is expressed in the DFS code with m 5-tuples. Therefore, for the database D including n graphs, the required memory space is at most $O(\Sigma_{i=1}^{n}|E(g_i)|)$. When adding a new graph to a code tree, if there are no nodes n for which $s(n)$ is identical to the code of the graph, the number of nodes in the code tree increases by $|E(g)|$, because the number of nodes added is the same as the number of edges in g. Therefore, the number of nodes in the code tree for the database D including n graphs is at most $O(\Sigma_{i=1}^{n}|E(g_i)|)$. Therefore, Lemma 5 has been proved.

Therefore, the size of the code tree becomes compact compared with storing all mappings between graphs in the database and patterns in the index. The complexity of constructing the code tree is $O(\Sigma_{i=1}^{n}|E(g_i)|)$, because an arbitrary code for each graph g is generated in $O(|E(g)|)$. When the average number of edges in the database is $\overline{E}$, its computational complexity is $O(\overline{E}|G|)$.

Next, we discuss the insertion, deletion, and update of database elements. When the graph g_{n+1} is inserted in G, there is no need to reconstruct the existing index, and only the loop statement of Algorithm 1 is executed. When deleting graph g_i from G, i is excluded from the ID list for node n including i. Finally, when updating g_i in G, after deleting g_i, the updated graph g_i is inserted. With this proposed method, it is not necessary to enumerate frequent subgraph patterns in G when inserting, deleting, or updating graphs in the database. Conversely, with conventional methods such as LW-index that use frequent patterns, when the graph database G becomes G' following insertion, deletion, and/or update operations, the frequent subgraph patterns enumerated from G may be very different to those enumerated from G'. In this case, it is necessary to reconstruct the index after obtaining frequent subgraph patterns from G'.

3.3 Supergraph Search Using Code Tree

Next, we propose a method for searching graphs in G that

is applicable to graph codes such as AcGM, exAcGM, and DFS. The proposed method assigns vertex IDs to vertices in q in accordance with the code definitions in parallel with traversing the code tree. Algorithm 3 is called by $search(\emptyset, q, \top, \langle\rangle)$ and returns the set of IDs of graphs that are subgraphs of q. In Line 2, we obtain the set of code fragments c that can be connected to the end of $code(q, w1, w2, \cdots, w_h)$. Here, for all elements $(w, c) \in C$, $c_1c_2\cdots c_hc$ is the prefix of one of possible codes for q. Next, for each pair $(m, (w, c))$ of children of node n and elements in C, Algorithm 4 or 5 is called on Line 5.

First, we consider the case in which graphs are expressed in the AcGM code. With the function *compare*, we check whether a subgraph in multiple graphs in G is a subgraph of q according to Lemma 1. Conditions (1) and (2) in Lemma 1 correspond to

(1) $\ell(v_i) = \ell(v'_i)$ and
(2) $\forall j < i\ (x_{j,i} = x'_{j,i} \vee x_{j,i} = 0)$

in Line 1 of Algorithm 3, respectively†. If Algorithm 4 always returns a value of "false" at node m, graphs whose IDs are in $ID(m')$ for all descendants m' of m are not subgraphs of q. Thus, it is possible to prune the search space by backtracking to this node m, which corresponds to filtering. Furthermore, because the subgraph isomorphic calculation in Algorithm 3 is shared between prefixes of multiple graphs in G and q, it is possible to further reduce the total computation time compared with applying the subgraph isomorphic test between the prefixes of each graph in G and q. Conversely, if Algorithm 4 returns a value of "true" at node m, graphs whose IDs are $ID(m)$ are subgraphs of q, which corresponds to verification. Therefore, in the proposed algorithm, distinguishing between filtering and verification allows them to be computed in the same algorithm rather than through individual, specific algorithms.

Algorithm 3: search

Data: a set of graph IDs S, the query q, the current node n, and argument $\langle w_1, w_2, \cdots w_h\rangle$ to create a code from q
Result: a set of graph IDs S

1 $S \leftarrow S \cup ID(n)$;
2 $C \leftarrow \{(w, c) \mid c_1c_2\cdots c_hc = code(q, \langle w_1, w_2, \cdots w_h, w\rangle), c_1c_2\cdots c_hc \subseteq s,\ s \in \Omega(q)\}$;
3 $N \leftarrow children(n)$;
4 **for** $(m, (w, c)) \in N \times C$ **do**
5 **if** $compare(fr(m), c)$ **then**
6 $S \leftarrow search(S, q, m, \langle w_1, w_2, \cdots w_h, w\rangle)$;
7 **return** S;

Algorithm 4: "*compare*" function for AcGM code

Data: fragments $\ell(v_i)x_{1,i}x_{2,i}\cdots x_{i-1,i}$ for $g \in G$ and $\ell(v'_i)x'_{1,i}x'_{2,i}\cdots x'_{i-1,i}$ for query q
Result: true or false

1 **if** $\ell(v_i) = \ell(v'_i) \wedge \forall j < i\ (x_{j,i} = x'_{j,i} \vee x_{j,i} = 0)$ **then**
2 **return** true;
else
3 **return** false;

Algorithm 5: "*compare*" function for DFS code

Data: fragments c_1 and c_2
Result: true or false

1 **if** $c_1 = c_2$ **then**
2 **return** true;
else
3 **return** false;

†When graphs in the database and queries are expressed in the exAcGM code, condition (3) in Lemma 2 is added to Line 1 of Algorithm 4.

For example, we consider $search(\emptyset, q, n_6, \langle v_1, v_2, v_3\rangle)$ to be called for a query q, shown in Fig. 5, at node n_6 in Fig. 4. As a subgraph of q induced by a set of vertices generated by adding either v_4 or v_5 to $\{v_1, v_2, v_3\}$ is connected, C in Line 2 of Algorithm 3 is $\{(v_4, a00), (v_5, 00a)\}$. In addition, as n_7 has a child, $C = \{n_7\}$. Thus, *compare* is called with the arguments $(n_7, (v_4, a00))$ and $\{(n_7, (v_5, 00a))$, and it returns "true" when called with $(n_7, (v_4, a00))$.

The code tree does not store any mappings between graphs in the database and patterns in the tree. In addition, Algorithm 3 does not create any mappings between the query and the patterns. Instead, Algorithm 3 traverses the code tree in a similar manner to the depth-first search. Different from the depth-first search, it may visit each node in the code tree multiple times, because a graph represented by a node may be isomorphic to subgraphs consisting of different subsets of vertices in q. For example, when the query in Fig. 5 is given for the code tree in Fig. 4, Algorithm 3 visits its node n_6 six times, as the permutation of v_1, v_2, and v_3 in q generates an isomorphic graph. When subgraphs of the given query are automorphisms, the number of mappings in the conventional methods and the frequency of visits in the proposed method become huge. In Sect. 3.4.2, we discuss an optimization method that reduces the frequency of visits to some nodes in the code tree, making the proposed method much more efficient.

Given a query q, if there is a graph g in DB which is a subgraph of q, Algorithm 3 can find the graph g from code tree, whose reasons are as follows.

(1) As mentioned after Proof 3, when $g \in G$ is a subgraph of the query q, there must be at least one code for q that satisfies one of the Lemmas 1, 2 and 3 with respect to the code for g.

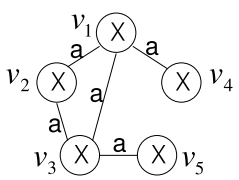

Fig. 5 A query graph.

(2) One of all possible codes of each graph $g \in G$ is contained in the code tree.
(3) Algorithm 3 generates all possible codes of the query q.

Until Sect. 3.3, we do not suggest that the proposed method is faster than LW-index which stores all mappings between g and q, which is equivalent with (3). However, since we avoid generating all the codes of q by using the optimized algorithm based on Algorithm 3, that is mentioned in Sect. 3.4, our proposed method becomes faster than LW-index.

Next, we consider the case in which graphs are expressed in the DFS code. If $fr(m)$ and c are the same as in Line 5 of Algorithm 3, then Algorithm 3 is called recursively. In the same way, using the DFS code has the effect of pruning the search space and sharing the subgraph isomorphic calculation, as when using the AcGM code.

3.4 Optimizations

3.4.1 Using Canonical Code

In the code tree T_1 in Fig. 4, $s(n_4)$ and $s(n_8)$ represent an isomorphic graph. If the nodes become a single node, we can reduce both the size of the code tree and the computation time of searching. To do so, we use canonical codes to represent graphs in the database [9], [17].

Definition 7 (canonical code): There are multiple AcGM codes representing an isomorphic graph g. We call the maximum among these codes the canonical code of g. In the same way, there are multiple DFS codes representing an isomorphic graph g. Among these, the minimum is called the canonical code of g.

The procedure for obtaining an arbitrary code of a graph g on Line 5 of Algorithm 1 is replaced by the procedure for obtaining the canonical code of g. The computational complexity of obtaining the canonical code for a graph is the same as the complexity of solving the graph isomorphic problem [6], and the computation time required to find canonical codes for all graphs in the database will be large. To reduce the computation time, a suboptimal canonical code (subcanonical code) for each graph is obtained using the breadth-first search, and this search is restricted by a certain beam width b.

Lemma 6: The time complexity for obtaining a subcanonical code of a graph g is $O(b|V(g)|^3)$ for the AcGM code and $O(b|E(g)||V(g)|)$ for the DFS code.

Proof 6: Algorithm 6 shows the pseudocode for obtaining a subcanonical code for the AcGM code for a given graph g and a beam width b. Here, $select(X, x)$ is the function for selecting x codes from a set of codes X. In Algorithm 6, Lines 3–10 are repeated $|V(g)|$ times and Lines 5–6 are repeated at most b times. In Line 6, the number of candidate vertices of u is at most $|V(g)|$. In addition, the time complexity for appending a fragment to $code(g, \langle v_1, v_2, \cdots, v_{d-1}\rangle)$ is

Algorithm 6: Obtaining a subcanonical code for an AcGM code

```
   Data: a graph g and a beam width b
   Result: a subcanonical code c
1  P ← {code(g, ⟨⟩)};
2  d ← 1;
3  for d ∈ [1, |V(g)|] do
4      P' ← φ;
5      for code(g, ⟨v1, v2, ··· , vd−1⟩) ∈ P do
6          P' ← P' ∪ {code(g, ⟨v1, v2, ··· , vd−1, u⟩) | u ∈
             V(g), u ∉ ⟨v1, v2, ··· , vd−1⟩};
7      m ← max P';
8      P ← {p ∈ P' | p = m};
9      P ← select(P, b);
10     d ← d + 1;
11 c ← select(P, 1);
12 return c;
```

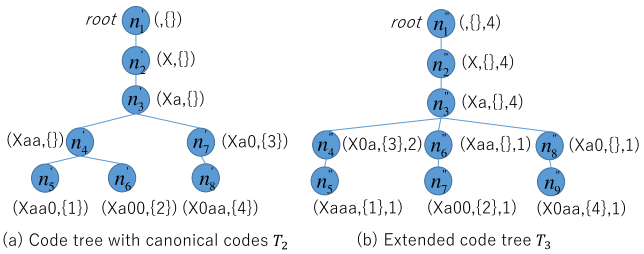

Fig. 6 Code trees.

$O(|V(g)|)$. Therefore, the time complexity for obtaining a subcanonical code of a graph g is $O(b|V(g)|^3)$ for the AcGM code. The subcanonical DFS code for a graph g is obtained in a similar manner to Algorithm 6. The differences between obtaining subcanonical codes for AcGM and DFS codes are in Lines 3 and 6. For a DFS code, Lines 3–10 are repeated $|E(g)|$ times, and the time complexity of Line 6 for appending a fragment to the code is $O(1)$. Therefore, the time complexity for obtaining a subcanonical code of a graph g is $O(b|E(g)||V(g)|)$ for the DFS code. From the above, Lemma 6 has been proved.

Using canonical codes, we have three aspects that accelerate the computation of our proposed method. (1) Fig. 6 (a) shows the code tree T_2 with canonical codes for the same database shown in Fig. 4. Compared with code tree T_1 in Fig. 4, T_2 has fewer nodes because there are fewer corresponding to each isomorphic graph in T_1. In Algorithm 3, the computation of whether graphs represented by $s(n_4)$ and $s(n_8)$ are subgraphs of the query are conducted simultaneously for n'_7 in T_2, which makes our proposed method more efficient. (2) Prefixes consisting of three vertices in g_1 of Fig. 4 are $s(n_4)$ in T_1 and $s(n'_4)$ in T_2. The graphs represented by $s(n_4)$ and $s(n'_4)$ have two and three edges, respectively. Using canonical codes, dense subgraphs of graphs in the database are arranged in the upper part of the code tree (root side). Compared with sparse subgraphs, the dense subgraphs are infrequent in G according to the definition of the support value. When the proposed method checks

whether the query contains g_1, the use of dense subgraphs in g_1 is an effective filtering method. Therefore, our proposed method uses infrequent subgraph patterns for filtering rather than frequent subgraph patterns. (3) For the AcGM code, we assign large values to the rare vertex labels and edge labels[†]. For example, in the next section, we use graphs of chemical compounds to evaluate the proposed method, and atom types in these chemical compounds correspond to vertex labels. Carbon and nitrogen are included in almost all chemical compounds, whereas holmium, terbium, and so on are rare. As the canonical code of a graph is the maximum among codes representing the graph, vertex labels having large values often appear in the early part (left-hand side) of the canonical code. This means that prefixes consisting of the early part of the canonical code represent infrequent subgraphs in G. Similar to (2), the effect of filtering in the proposed method is further accelerated.

Conversely, with conventional methods such as the LW-index, some frequent subgraph patterns in G are used for the index. The frequent subgraph patterns have a high possibility of being included in the query q, and if a pattern p in the index is included in q, graphs in G including p cannot be pruned from the search space. The use of infrequent subgraph patterns in the code tree is a different characteristic from conventional methods, which typically use frequent subgraph patterns.

3.4.2 Storing the Numbers of Candidate Solutions

As mentioned in Sect. 3.3, for the query q in Fig. 5, our proposed method visits n_6 in T_1 six times. Similarly, many conventional methods store six mappings between the pattern represented by n_6 and q, but visit the node only once. In a graph search, these procedures are very time-consuming. To avoid multiple visits when using the proposed method, we extend the aforementioned code tree and Algorithm 3 as follows. Each node n in the code tree has a code fragment, a set of graph IDs, and an integer $num(n)$. The integer stores the summation of cardinalities of graph IDs in a subtree $T_s(n)$ whose root is n. Figure 6 (b) shows the extended code tree T_3 for code tree T_1 and Algorithm 7 is the extension of Algorithm 3. When Algorithm 7 first visits n_7'' in T_3 after visiting n_3'' and n_6'', it adds $\{1\}$ to S and decrements num values of nodes on the path from the root to n_7'', in Lines 3–5. When Algorithm 7 visits n_3'' again, n_6'' is removed from N in Line 7, because the subtree whose root is n_6'' does not have any solutions at this point in time. Pruning n_6'' from T_3 reduces the number of times that Algorithm 7 visits the same nodes, which further accelerates the proposed method. Finally, all num values in T_3 are reset at the end of the search for the given query in Lines 11–12.

4. Experimental Evaluation

We implemented the method proposed in this paper and the conventional LW-index method in Java, and performed an experimental evaluation. A previous comparison [21] found that the LW-index is faster than other methods [2], [23], [25] for a supergraph search. The experiments reported in this section were performed on a workstation with a Xeon X5670 293 GHz CPU and 48 GB main memory. Furthermore, the dataset used in our experiments includes 39,338 chemical compounds [12], similar to that considered in [21], and the atoms, bonds, atom types, and bond types in each chemical compound were treated as vertices, edges, vertex labels, and edge labels, respectively. We denote the set of graphs as G_{aids}. Moreover, α graphs were chosen at random from G_{aids} and set to G_α. Furthermore, we randomly extracted a set of frequency subgraph patterns $F_{0.5}$ from G_{aids} with a support threshold of 0.5%. Then, β graphs were extracted at random from $F_{0.5}$ and set to G'_β. Two types of query sets, Q_1 and Q_2, were generated as subsets of G_{aids}. Q_1 was formed from the graphs within G_{aids} that have 25 edges, and Q_2 was the set of graphs with 34–36 edges; $|Q_1|$ = 1,835 and $|Q_2|$ = 2,223. Table 2 summarizes the data features used in our experiments.

[†]Conversely, for the DFS code, we assign small values to the rare labels in accordance with the definition of the canonical code.

Algorithm 7: Search 2

```
   Data: Arguments are the same as in Algorithm 3
   Result: a set of graph IDs S
1  S ← S ∪ ID(n);
2  m ← n;
3  while m ≠ null do
4      num(m) ← num(m) − |ID(n)|;
5      m ← parent(m);
6  C ← {(w, c) | c1c2 ··· chc = code(q, ⟨w1, w2, ··· wh, w⟩),
           c1c2 ··· chc ⊆ s, s ∈ Ω(q)};
7  N ← {m | m ∈ children(n), num(m) ≠ 0};
8  for (m, (w, c)) ∈ N × C do
9      if compare(fr(m), c) then
10         S ← search(S , q, m, ⟨w1, w2, ··· wh, w⟩);
11 if n is root then
12     reset all num values in this code tree;
13 return S;
```

Table 2 Features of evaluation data G_{aids}.

No. of graphs	39,338
No. of vertex labels	59
No. of edge labels	3
Average no. of vertices	25.15
Average degree	2.16
Max no. of vertices	222
Max no. of edges	234

4.1 Using Canonical Codes

4.1.1 Indexing

Figures 7, 8, and 9 show the performance of the proposed method when various beam widths are used to find sub-canonical codes. In these experiments, the graph database was G_{aids} and the queries were Q_1 and Q_2. Figure 7 shows

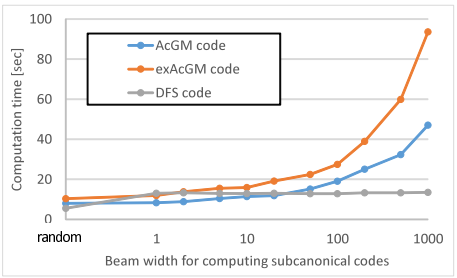

Fig. 7 Computation time for constructing code trees with respect to beam width.

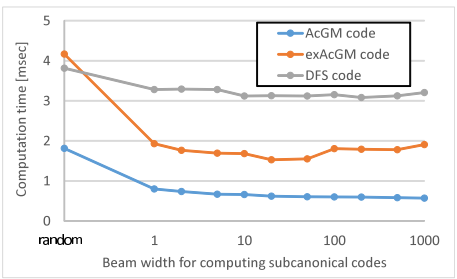

Fig. 8 Computation time required to search for graphs in Q_1.

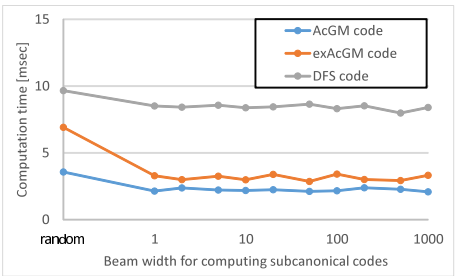

Fig. 9 Computation time required to search for graphs in Q_2.

the computation time required to create code trees (indexes) when changing the beam width. The "random" marker on the horizontal axis refers to results without using any (sub)canonical codes. For AcGM-based codes, the calculation time increases with the beam width, because *getCode* in Algorithm 1 calculates a subcanonical code for each graph in G_{aids}. Conversely, for the DFS code, the computation time is almost constant. This is because the graphs for the chemical compounds are sparse. When a graph is traversed depth-first to generate DFS codes for the graph, there are limited vertices to be visited after a certain vertex, which reduces the variation in the DFS codes of the graph and restricts the time required to construct the index.

4.1.2 Searching

Figure 8 shows the average computation time required to search for the graphs in Q_1. When the beam width (the parameter for constructing the code tree) is increased, the computation time decreases for the AcGM, exAcGM, and DFS codes. This confirms the positive effect of using

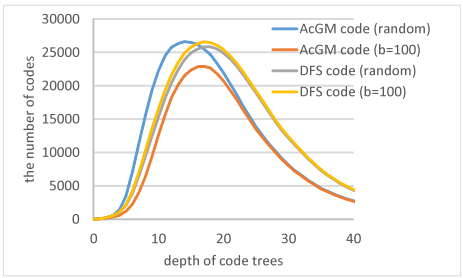

Fig. 10 Size of the code trees.

Table 3 Average computation time [msec] of Algorithms 3 and 7 for queries Q_1 and Q_2.

	Q_1		Q_2	
	Algo. 3	Algo. 7	Algo. 3	Algo. 7
AcGM	0.60	0.55	20.67	2.18
exAcGM	2.10	1.88	37.53	3.17
DFS	3.64	3.09	357.49	8.30

(sub)canonical codes. Figure 9 shows the average computation time required to search for the graphs in Q_2. We obtained a similar result as in Fig. 8.

The computation time using the AcGM code is shorter than that using the DFS code. To explain this, the relationship between the depths and the numbers of nodes in the code trees is shown in Fig. 10. When seeking subcanonical codes of AcGM codes with a beam width b of 100, the number of nodes in the code tree is low compared to one for *random* AcGM codes This means that the prefixes of AcGM codes representing graphs in G_{aids} are held on common branches in the code tree. As the prefixes are held on common branches, whether q includes *many* graphs represented by these prefixes is calculated concurrently, and the AcGM code enables us to search the graphs in less time. As the above results confirm the positive effect of using subcanonical codes, subsequent experiments were conducted with the beam width fixed at 100.

4.2 Storing the Numbers of Candidate Solutions

Table 3 presents the average computation time using Algorithms 3 and 7, as explained in Sect. 3. For the DFS code applied to Q_2, the computation time of Algorithm 3 was reduced by approximately 97.5% using Algorithm 7. Table 4 lists the numbers of nodes in code trees visited by Algorithms 3 and 7 for each query in Q_1 and Q_2. Algorithm 7 clearly visits fewer nodes than Algorithm 3. In our code trees, the values $num(n)$ store the numbers of candidate solutions in the subtree for each node whose root is n. In Algorithm 7, the value for each node changes dynamically as the code tree is traversed. If $num(n) = 0$ at node n, the subtree for n does not need to be traversed. For this reason, Algorithm 7 visits fewer nodes than Algorithm 3, which results in a lower computation time. The effect of storing the numbers of candidate solutions is profound for Q_2 because this set includes many automorphisms, such as the graph shown

Table 4 Numbers of nodes that Algorithms 3 and 7 visit per query in Q_1 and Q_2.

	Q_1		Q_2	
	Algo. 3	Algo. 7	Algo. 3	Algo. 7
AcGM	584.7	507.1	54,745.8	2,442.7
exAcGM	797.3	722.5	44,003.2	1,346.3
DFS	747.5	676.9	56,281.7	1,364.9

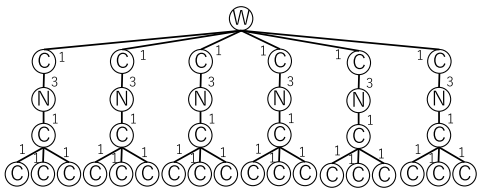

Fig. 11 A query in Q_2 (numbers are edge labels).

in Fig. 11. Given such a graph as a query, Algorithm 7 does not visit the nodes corresponding to the graphs in the query multiple times, which reduces the computation time.

4.3 Comparison with LW-Index Method

4.3.1 Setting

We now compare the proposed method with the LW-index method[†]. The experimental setting is the same as that described in [21], the graph database is G'_β, and the queries are Q_1 or Q_2. To construct the LW-index, a complete set of frequent subgraph patterns are enumerated with a minimum support level of 1% from G'_{100000}. A small subset of the frequent subgraph patterns is chosen when constructing the LW-index, and the chosen graphs are used for filtering during the search. When constructing the LW-index, graphs in Q_1 or Q_2 were used as queries. Generally, as the queries are unknown when constructing the index in advance of searching, Q_1 or Q_2 would not usually be used to construct the index, but this experiment placed the LW-index method in a clearly advantageous position to demonstrate the superiority of the proposed method.

4.3.2 Results

Figure 12 shows the average computation time required to search for graphs in Q_1 when changing the number β of graphs in the database. In the same way, Fig. 13 shows the experimental results with Q_2. As β increases, the computation time increases significantly, although that of the proposed method is far lower than that of the LW-index method for β above 10,000. The reason for this is as follows. We consider the case in which one graph g'_1 is newly inserted in G'_β and the canonical code of g'_1 is the prefix of the canonical code of another graph g'_2 in G'_β. In this case, the code trees for G'_β and $G'_\beta \cup \{g'_1\}$ are identical, and the nodes traversed by Algorithm 7 do not change. As the database G'_β

[†]Henceforth, we refer to the methodology of the LW-index and the index itself as the "LW-index method" and "LW-index," respectively, to distinguish them.

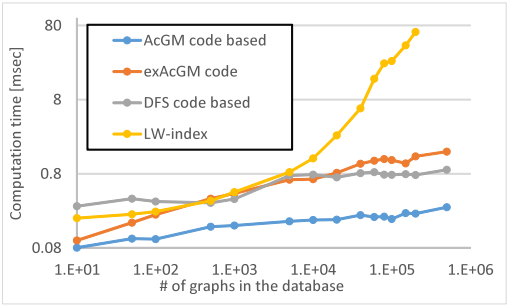

Fig. 12 Comparison with LW-index (G'_β, Q_1).

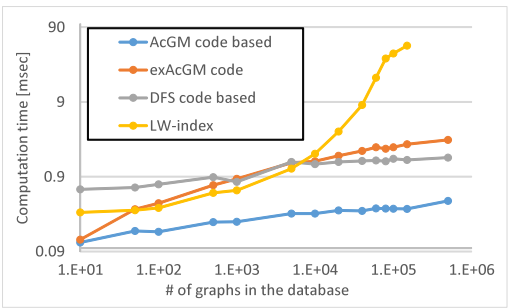

Fig. 13 Comparison with LW-index (G'_β, Q_2).

used in this experiment is a subset of $F_{0.5}$, as β increases, the number of graphs that are newly inserted in the database and whose codes are prefixes of codes for graphs already in the database increases. Therefore, even when the number of graphs in the database increases, the computation time for the proposed method does not increase as quickly as that of the LW-index method. Indeed, the LW-index method solves the subgraph isomorphic problem (verifications) between the query and graphs that could not be excluded by filtering. As the size of G'_β increases, the number of verifications increases, so the calculation time for the LW-index method increases with β.

4.4 Comparison with LW-Index Method with Large Graphs

4.4.1 Setting

The graphs in G'_β had an average of 13.67 vertices, which is not very large. Therefore, we also compared the performance of the proposed method with that of the LW-index method for a dataset consisting of graphs with a large number of vertices. In this experiment, the graph database was G_α and the queries were Q_1 or Q_2. To construct the LW-index, frequent subgraph patterns were enumerated with a minimum support threshold of 1% from G_{aids}. Furthermore, Q_1 and Q_2 were used when constructing the LW-index.

4.4.2 Results

Figure 14 shows the average computation time required to

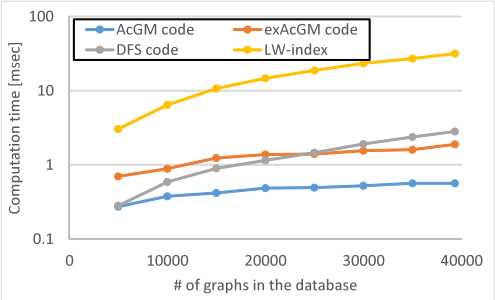

Fig. 14 Comparison with LW-index (G_α, Q_1).

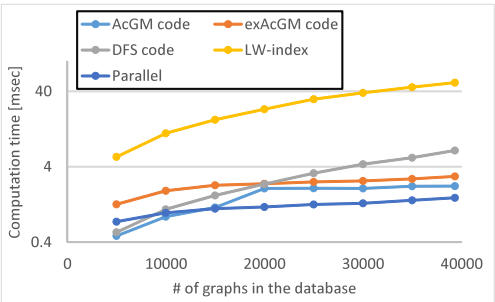

Fig. 15 Comparison with LW-index (G_α, Q_2).

search for graphs in Q_1 with respect to the number α of graphs in the database. In the same way, Fig. 15 shows the experimental results for Q_2[††]. Compared with the experiments described in Sect. 4.3, the graphs in G_{aids} are large and relatively few graphs are subgraphs of others in the database. For this type of database, the proposed method is much more efficient than the LW-index method. Furthermore, as shown in Fig. 7, the computation time required for constructing a code tree when using the AcGM code is approximately 20 seconds. In addition, the code trees for Q_1 and Q_2 are identical. Conversely, with the conventional method, it is necessary to enumerate frequent subgraph patterns from G to create the index; even using the most efficient frequent subgraph pattern mining method, Gaston [15], this requires approximately 400 seconds. If we chose filtering patterns from the 105,418 frequent subgraph patterns, the computation time required to construct the index would be enormous. With the LW-index method, the indices for Q_1 and Q_2 are different, and it is necessary to reconstruct the index every time the distribution of the search queries changes. Therefore, the proposed method is superior for both index construction and searching.

4.5 Parallel Computation on Two Different Indexes

In Fig. 16, each point (t, t_{ex}) gives the computation times t and t_{ex} for each query in Q_2 using the AcGM and exAcGM codes, respectively. The gravity of all points in Fig. 16 is marked by the red point. There are many points above the red line, which suggests that the computation time using the AcGM code is shorter than when using the exAcGM code for many graphs in G_{aids}. However, for a few graphs in Q_2, the proposed method using the AcGM code has a longer computation time than when using the exAcGM code. Figure 11 shows one of the graphs, which is an automorphism. To utilize the advantageous characteristics of the codes, we implemented a method in which two threads search in parallel for graphs in the database. In this method, two code trees are stored in memory, one in which the graphs are expressed in AcGM code and another in which the graphs are expressed in exAcGM code. For a given query, one of the threads traverses the former code tree and the other thread traverses the latter. When either thread finishes searching for the query, it forces the other thread to terminate the search. The computation time using this method is denoted as "Parallel" in Fig. 15. The average computation time of Parallel is shorter than that of the methods using the AcGM and exAcGM codes.

[††]The result marked "Parallel" in Fig. 15 is explained in Sect. 4.5.

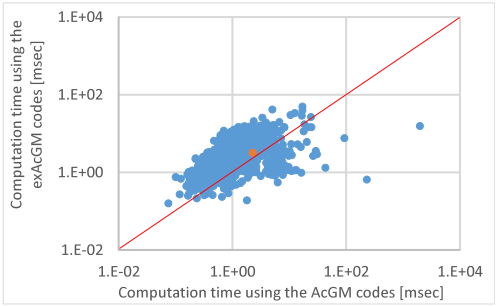

Fig. 16 Detailed computation time for each query in Q_2.

5. Conclusion

In this paper, we proposed a graph search method in which the search index does not require any knowledge of the query set or the frequent subgraph patterns. In conventional techniques, enumerating and selecting frequent subgraph patterns is computationally expensive, and the distribution of the query set must be known in advance. Subsequent changes to the query set require the frequent patterns to be selected again and the index to be reconstructed. The proposed method overcame these difficulties through graph coding, using a tree structured index that contains infrequent subgraph patterns in the shallow part of the tree. By traversing this code tree, the proposed method can rapidly determine whether multiple graphs in the database contain subgraphs that match the query, producing a powerful pruning or filtering effect. Furthermore, the filtering and verification steps of the graph search can be conducted concurrently, rather than requiring separate algorithms. As the proposed method does not require the frequent subgraph patterns and the query set, it is significantly faster than previous techniques; this independence from the query set also means that there is no need to reconstruct the search index when the query set changes. A series of experiments using a real-

world dataset demonstrated the efficiency of the proposed method, achieving a search speed several orders of magnitude faster than the previous best. In future work, we plan to compare DGTree [13] which also does not require any knowledge of the query set or the frequent subgraph patterns to construct its index but stores all mappings between graphs in the database and patterns in the index.

References

[1] M. Cannataro, P.H. Guzzi, and P. Veltri, "Protein-to-protein interactions: Technologies, databases, and algorithms," ACM Computing Surveys, vol.43, no.1, Article 1, 2010.
[2] C. Chen, X. Yan, P.S. Yu, J. Han, D.-Q. Zhang, and X. Gu, "Towards graph containment search and indexing," Proc. International Conference on Very Large Data Bases (VLDB), pp.926–937, 2007.
[3] J. Cheng, Y. Ke, W. Ng, and A. Lu, "Fg-index: Towards verification-free query processing on graph databases," Proc. ACM SIGMOD International Conference on Management of Data, pp.857–872, 2007.
[4] J. Cheng, Y. Ke, A.W.-C. Fu, and J.X. Yu, "Fast graph query processing with a low-cost index," The VLDB Journal, vol.20, no.4, pp.521–539, 2011.
[5] S. Fortin, "The graph isomorphism problem," Technical Report TR96-20, Department of Computer Science, University of Alberta, 1996.
[6] M.R. Garey and D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman, 1979.
[7] W.-S. Han, J. Lee, and J.-H. Lee, "Turbo$_{iso}$: Towards ultrafast and robust subgraph isomorphism search in large graph databases," Proc. ACM SIGMOD International Conference on Management of Data, pp.337–348, 2013.
[8] H. He and A.K. Singh, "Query language and access methods for graph databases," Managing and Mining Graph Data, Advances in Database Systems, vol.40, pp.125–160, Springer, Boston, MA, 2010.
[9] A. Inokuchi, T. Washio, and H. Motoda, "An apriori-based algorithm for mining frequent substructures from graph data," Proc. European Conference on Principles of Data Mining and Knowledge Discovery, Lecture Notes in Computer Science, vol.1910, pp.13–23, Springer, Berlin, Heidelberg, 2000.
[10] A. Inokuchi, T. Washio, Y. Nishimura, and H. Motoda, "A fast algorithm for mining frequent connected subgraphs," IBM Research, Tokyo Research Laboratory, 2002.
[11] H. Jiang, H. Wang, P.S. Yu, and S. Zhou, "GString: A novel approach for efficient search in graph databases," Proc. International Conference on Data Engineering, pp.566–575, 2007.
[12] S. Kramer, L. De Raedt, and C. Helma, "Molecular feature mining in HIV data," Proc. International Conference on Knowledge Discovery and Data Mining, pp.136–143, 2001.
[13] B. Lyu, L. Qin, X. Lin, L. Chang, and J.X. Yu, "Scalable supergraph search in large graph databases," Proc. IEEE International Conference on Data Engineering, pp.157–168, 2016.
[14] L. Ma, Z. Huang, and Y. Wang, "Automatic discovery of common design structures in CAD models," Computers & Graphics, vol.34, no.5, pp.545–555, 2010.
[15] S. Nijssen and J.N. Kok, "A quickstart in frequent structure mining can make a difference," Proc. International Conference on Knowledge Discovery and Data Mining, pp.647–652, 2004.
[16] S. Nowozin, K. Tsuda, T. Uno, T. Kudo, and G. Bakir, "Weighted substructure mining for image analysis," Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2007.
[17] H. Shang, Y. Zhang, X. Lin, and J.X. Yu, "Taming verification hardness: An efficient algorithm for testing subgraph isomorphism," Proc. VLDB Endowment, vol.1, no.1, pp.364–375, 2008.
[18] A. Trémeau and P. Colantoni, "Regions adjacency graph applied to color image segmentation," IEEE Trans. Image Process., vol.9, no.4, pp.735–744, 2000.
[19] W.T. Wipke and D. Rogers, "Artificial intelligence in organic synthesis. SST: Starting material selection strategies. An application of superstructure search," Journal of Chemical Information and Computer Sciences, vol.24, no.2, pp.71–81, 1984.
[20] X. Yan and J. Han, "gSpan: Graph-based substructure pattern mining," Proc. IEEE International Conference on Data Mining, pp.721–724, 2002.
[21] D. Yuan, P. Mitra, and C.L. Giles, "Mining and indexing graphs for supergraph search," Proc. VLDB Endowment, vol.6, no.10, pp.829–840, 2013.
[22] S. Zhang, M. Hu, and J. Yang, "TreePi: A novel graph indexing method," Proc. International Conference on Data Engineering, pp.966–975, 2007.
[23] S. Zhang, J. Li, H. Gao, and Z. Zou, "A novel approach for efficient supergraph query processing on graph databases," Proc. International Conference on Extending Database Technology, pp.204–215, 2009.
[24] P. Zhao, J.X. Yu, and P.S. Yu, "Graph Indexing: Tree + Delta >= Graph," Proc. International Conference on Very Large Data Bases (VLDB), pp.938–949, 2007.
[25] G. Zhu, X. Lin, W. Zhang, W. Wang, and H. Shang, "PrefIndex: An efficient supergraph containment search technique," Proc. International Conference Scientific and Statistical Database Management, Lecture Notes in Computer Science, vol.6187, pp.360–378, Springer, Berlin, Heidelberg, 2010.
[26] Q. Zhu, J. Yao, S. Yuan, F. Li, H, Chen, W. Cai, and Q. Liao, "Superstructure searching algorithm for generic reaction retrieval," Journal of Chemical Information and Modeling, vol.45, no.5, pp.1214–1222, 2005.
[27] X. Yan, P.S. Yu, and J. Han, "Graph indexing: A frequent structure-based approach," SIGMOD Conference, pp.335–3462, 2004.

Shun Imai received the B.S. degree in science and technology from Kwansei Gakuin University, Japan, in 2018. He is a master course student in School of Science and Technology, Kwansei Gakuin University. He works on the study of data mining and data engineering. School of Science and Technology, Kwansei Gakuin University, 2-1 Gakuen, Sanda, Hyogo, 669-1337 Japan.

Akihiro Inokuchi received the Ph.D. degree in communication engineering from Osaka University, Japan, in 2004. He is a professor in School of Science and Technology, Kwansei Gakuin University. He works on the study of data mining, machine learning, artificial intelligence and data engineering. He received the best paper award from the Journal Award of Computer Aided Chemistry in 2002, and the incentive awards from Japanese Society for Artificial Intelligence in 2004 and 2008. School of Science and Technology, Kwansei Gakuin University, 2-1 Gakuen, Sanda, Hyogo, 669-1337 Japan.