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 chemoinformatics [19], [26], bioinformatics [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 \( G = \{g_1, g_2, \ldots, g_p\} \) 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 \( \ell : V \cup E \rightarrow L \) 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) \( \phi : V' \rightarrow V \) that satisfies the following three conditions \( \forall v, v_1, v_2 \in V' \):

1. \( \ell'(v) = \ell(\phi(v)) \).
2. \( (\phi(v_1), \phi(v_2)) \in E, \text{ if } (v_1, v_2) \in E' \).
3. \( \ell'(\phi(v_1), \phi(v_2)) = \ell((\phi(v_1), \phi(v_2))) \).

The subgraph isomorphism problem is the NP-complete problem of finding mappings between \( g \) and \( g' \). The set \( \Phi \) of all possible permutations of integers \( 1, 2, \ldots, |V| \) 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

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

Furthermore, given a certain threshold \( \sigma' \), 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 \( p \not\subseteq g_i \).
- 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

\[
G' = G - \bigcup_{p \in P}\{g \mid g \in G, p \subseteq g\}
\]
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 \( v_1, v_2, \ldots, v_{|V|} \). \( g \) 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): If the AcGM codes $\alpha = a_1a_2\cdots a_k$ and $\beta = b_1b_2\cdots b_l$ are such that $a_i = e_{ij}x_{i,j} \cdots x_{i-1,j}$ and $b_j = \ell_jx'_{j,1} \cdots x'_{j,i-1,j}$ satisfy the following conditions for all $i \leq k$, the graph represented by $\alpha$ is a subgraph of the graph represented by $\beta$.

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

Proof 1: Let $g$ and $g'$ be the graphs expressed by $\alpha$ and $\beta$, 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 for a function $\phi$ where $\phi(i) = i$. In addition, $x_{ji} = x'_{ji}$ or $x_{ji} = 0$ suggests that there is an edge between $v_i' \in g'$ and $v_j' \in g'$, which satisfies Definition 1 for a function $\phi$ where $\phi(i) = i$. Furthermore, $x_{ji} = x'_{ji}$ 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 for a function $\phi$ where $\phi(i) = i$. 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 $\phi$ 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): If the AcGM codes $\alpha = a_1a_2\cdots a_k$ and $\beta = b_1b_2\cdots b_l$ are such that $a_i = e_{ij}x_{i,j} \cdots x_{i-1,j}$ and $b_j = \ell_jx'_{j,1} \cdots x'_{j,i-1,j}$ satisfy the following conditions for all $i \leq k$, the graph represented by $\alpha$ is a subgraph of the graph represented by $\beta$.

(1) $\ell_i = \ell'_i$.  
(2) $x_{ji} = x'_{ji}$ or $x_{ji} = 0$ for all $j < i$.  
(3) $d_i \leq d'_i$ and $d_2 \leq d'_{2}$.  

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 \( e_{ab} = (v, u) \) was discovered before \( e_{ba} = (v', u') \). The DFS code for graph \( g \) is defined as

\[
\text{code}(g, (e_1, e_2, \ldots, e_{|E|})) = c_1c_2 \cdots c_{|E|},
\]

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, \( \gamma < \alpha < \beta \).

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

Proof 3: Let the DFS codes \( \alpha \) and \( \beta \) be \( \alpha = c_1c_2 \cdots c_k \) and \( \beta = c'_1c'_2 \cdots c'_k \), where

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

If \( \alpha \) is a prefix of \( \beta \), \( c_i = c'_i \) holds for all \( i \leq k \). Therefore, \( \ell(v_i) = \ell(v'_i), \ell(u_i) = \ell(u'_i), \text{ and } \ell(v_i, u_i) = \ell(v'_i, u'_i) \), which satisfies Definitions 1(1), (2-1), and (2-2) for a function \( \phi \) 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.

### 3.2 Indexing the Graph Database

**Definition 6 (code tree):** We define the code tree \( T \) as \((T, N, B)\), where \( T \in N \) 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 \( g_1 \in G \) 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 \( fr(T) = \text{null} \) and \( ID(T) = \emptyset \). 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

\[
\begin{align*}
\text{code}(g_1, (v_1, v_2, v_3, v_4)) &= XaXa0aXaaa, \\
\text{code}(g_2, (v_1, v_2, v_3, v_4)) &= XaXaaXa00, \\
\text{code}(g_3, (v_1, v_2, v_3, v_4)) &= XaXa0a, \text{ and} \\
\text{code}(g_4, (v_1, v_2, v_3, v_4)) &= XaXa00Xaa0a.
\end{align*}
\]

The code tree \( T \) 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 \( ID(n_5) = \{1\} \). If we concatenate the code fragments on each node from the root node \( T = n_1 \) to node \( n_5 \), we obtain \( XaXa0aXaaa \), 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 \( T \). 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

| Table 1 DFS codes for graphs in Fig. 3. |
|-----------------|-----------------|-----------------|-----------------|
| \( c_i \)       | \( \alpha \)    | \( \beta \)     | \( \gamma \)    |
| \((\text{Fig. 3 (b)})\) | \((\text{Fig. 3 (c)})\) | \((\text{Fig. 3 (d)})\) |
| \( c_1 \)       | \( 1, 2, X, X, a \) | \( 1, 2, X, X, d \) | \( 1, 2, X, X, a \) |
| \( c_2 \)       | \( 2, 3, X, X, b \) | \( 2, 3, X, X, b \) | \( 2, 3, X, X, a \) |
| \( c_3 \)       | \( 3, 1, X, X, a \) | \( 3, 4, X, X, a \) | \( 3, 1, X, X, b \) |
| \( c_4 \)       | \( 3, 4, X, X, c \) | \( 4, 2, X, X, a \) | \( 3, 4, X, X, d \) |
| \( c_5 \)       | \( 4, 2, X, X, b \) | \( 3, 5, X, X, c \) | \( 3, 5, X, X, b \) |
| \( c_6 \)       | \( 2, 5, X, X, d \) | \( 5, 2, X, X, b \) | \( 5, 1, X, X, c \) |

![Fig. 4 Code tree example.](image-url)
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 $c_i$; 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 $B_3$ 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 $\bar{E}$, its computational complexity is $O(\bar{E} |G|)$.

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
Algorithm 4: 

Algorithm 3: search 

Data: a set of graph IDs \( S \), the query \( q \), the current node \( n \), and argument \((w_1, w_2, \ldots, w_h)\) to create a code from \( q \) 
Result: a set of graph IDs \( S \) 

1. \( S \leftarrow S \cup ID(n) \) 
2. \( C \leftarrow \{ (w, c) \mid c_1c_2 \cdots c_h = \text{code}(q, w_1, w_2, \ldots, w_h) \} \), \( c_1c_2 \cdots c_h \leq s, s \in \Omega(q) \) 
3. \( N \leftarrow \text{children}(n) \) 
4. for \((m, (w, c)) \in N \times C \) do 
  5. if \( \text{compare}(f(m), c) \) then 
  6. \( S \leftarrow \text{search}(S, q, m, (w_1, w_2, \ldots, w_h, w)) \) 
7. return \( S \) 

Algorithm 4: \( \text{“compare”} \) function for AcGM code 

Data: fragments \( \ell(v_i) \lambda_{x_{i,1}, x_{i,2}, \ldots, x_{i,j_i}} \) for \( g \in G \) and \( \ell(v_j) x_{j,1}', x_{j,2}', \ldots, x_{j,1}'' \) for query \( q \) 
Result: true or false 

1. if \( \ell(v_i) = \ell(v_j) \wedge \forall j < i \ (x_{ji} = x_{ji}' \vee x_{ji} = 0) \) then 
  2. return true; 
else 
  3. return false; 

Algorithm 5: \( \text{“compare”} \) 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; 

1When 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.
(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 \( code(g, \{v_1, v_2, \cdots, v_{d-1}\}) \) is \( O(|V(g)|) \). 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 \( n_4 \) and \( n_8 \) are subgraphs of the query are conducted simultaneously for \( n_2 \) 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
Finally, all same nodes, which further accelerates the proposed method.

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\(^1\). 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 \( \text{num}(n) \). The integer stores the summation of cardinalities of graph IDs in a subtree \( T_3(n) \) 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 \( n''_6 \) in \( T_3 \) after visiting \( n''_6 \) and \( n''_6 \), it adds \( \{1\} \) to \( S \) and decrements \( \text{num} \) values of nodes on the path from the root to \( n''_6 \), in Lines 3–5. When Algorithm 7 visits \( n''_6 \) 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 \( \text{num} \) 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 \( G_{\text{ads}} \). Moreover, \( \alpha \) graphs were chosen at random from \( G_{\text{ads}} \) and set to \( G_\alpha \). Furthermore, we randomly extracted a set of frequency subgraph patterns \( F_{0.5} \) from \( G_{\text{ads}} \) with a support threshold of 0.5\%. Then, \( \beta \) graphs were extracted at random from \( F_{0.5} \) and set to \( G_{\beta} \). Two types of query sets, \( Q_1 \) and \( Q_2 \), were generated as subsets of \( G_{\text{ads}} \). \( Q_1 \) was formed from the graphs within \( G_{\text{ads}} \) 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.

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 \( G_{\text{ads}} \) and the queries were \( Q_1 \) and \( Q_2 \). Figure 7 shows

---

\(^1\)Conversely, for the DFS code, we assign small values to the rare labels in accordance with the definition of the canonical code.
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 (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 \( \text{num}(n) \) 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 \( \text{num}(n) = 0 \) 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 $Q_2$.

<table>
<thead>
<tr>
<th></th>
<th>$Q_1$</th>
<th>$Q_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Algo. 3</td>
<td>Algo. 7</td>
</tr>
<tr>
<td>AcGM</td>
<td>584.7</td>
<td>507.1</td>
</tr>
<tr>
<td>exAcGM</td>
<td>797.3</td>
<td>722.5</td>
</tr>
<tr>
<td>DFS</td>
<td>747.5</td>
<td>676.9</td>
</tr>
</tbody>
</table>

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 $G'_\beta$ 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 $\beta$ of graphs in the database. In the same way, Fig. 13 shows the experimental results with $Q_2$. As $\beta$ increases, the computation time increases significantly, although that of the proposed method is far lower than that of the LW-index method for $\beta$ above 10,000. The reason for this is as follows. We consider the case in which one graph $g'_1$ is newly inserted in $G'_\beta$ and the canonical code of $g'_1$ is the prefix of the canonical code of another graph $g'_2$ in $G'_\beta$. In this case, the code trees for $G'_\beta$ and $G'_\beta \cup \{g'_1\}$ are identical, and the nodes traversed by Algorithm 7 do not change. As the database $G'_\beta$ used in this experiment is a subset of $F_{0.5}$, as $\beta$ 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 $G'_\beta$ increases, the number of verifications increases, so the calculation time for the LW-index method increases with $\beta$.

4.4 Comparison with LW-Index Method with Large Graphs

4.4.1 Setting

The graphs in $G'_\beta$ 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 $G_\alpha$ 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 $G_\alpha_{ids}$. 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
search for graphs in $Q_1$ with respect to the number $\alpha$ of graphs in the database. In the same way, Fig. 15 shows the experimental results for $Q_2$.\textsuperscript{\textdagger}\textsuperscript{\textdagger}\textsuperscript{\textdagger} Compared with the experiments described in Sect. 4.3, the graphs in $G_{aid}$ 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\textsuperscript{[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 $G_{aid}$. 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 ex-AcGM codes.

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


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.

Akhiro 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.