algorithms

MDPI

Article

Similar Supergraph Search Based on Graph Edit Distance

Masataka Yamada and Akihiro Inokuchi *

Graduate School of Science and Technology, Kwansei Gakuin University, 2-1 Gakuen,
Sanda 669-1337, Hyogo, Japan; eyd13231@kwansei.ac.jp
* Correspondence: inokuchi@kwansei.ac.jp

Abstract: Subgraph and supergraph search methods are promising techniques for the development of new drugs. For example, the chemical structure of favipiravir—an antiviral treatment for influenza—resembles the structure of some components of RNA. Represented as graphs, such compounds are similar to a subgraph of favipiravir. However, the existing supergraph search methods can only discover compounds that match exactly. We propose a novel problem, called similar supergraph search, and design an efficient algorithm to solve it. The problem is to identify all graphs in a database that are similar to any subgraph of a query graph, where similarity is defined as edit distance. Our algorithm represents the set of candidate subgraphs by a code tree, which it uses to efficiently compute edit distance. With a distance threshold of zero, our algorithm is equivalent to an existing efficient algorithm for exact supergraph search. Our experiments show that the computation time increased exponentially as the distance threshold increased, but increased sublinearly with the number of graphs in the database.

Keywords: labeled graph; supergraph search; similarity graph; graph edit distance

check for updates

Citation: Yamada, M.; Inokuchi, A. Similar Supergraph Search Based on Graph Edit Distance. *Algorithms* **2021**, *14*, 225. https://doi.org/10.3390/a14080225

Academic Editors: Deepak Ajwani, Sabine Storandt and Darren Strash

Received: 22 June 2021
Accepted: 23 July 2021
Published: 27 July 2021

1. Introduction

The coronavirus disease (COVID-19) has been spreading widely since 2019. In Japan, favipiravir (brand name: Avigan) has been examined as a promising antiviral medication against COVID-19. Favipiravir was initially developed as a medication to treat influenza. When Favipiravir is ribosylated, its structure resembles acadesine, which is a precursor to inosine. Inosine is the previous stage of guanosine and adenosine, which are components of RNA [1]. With this treatment, the RNA synthetase of a virus mistakenly incorporates favipiravir, instead of guanosine, into the replicating virus. This causes RNA synthesis to stop and limits the replication of viruses inside the body.

Figure 1 shows the chemical structures of guanosine, inosine, acadesine, and ribosylated favipiravir. The substructure α, drawn in red, is common to all of guanosine, inosine, and acadesine. Therefore, in order to find candidate compounds for a new drug, it may be useful to search for compounds containing a substructure, provided as a query, in a database that consist of many compounds (i.e., substructure search). Conversely, if many substructures relevant to new drug development are known, it can be useful to search for substructures contained in a newly developed chemical compound, provided as a query, in a database recording these substructures (i.e., superstructure search). This paper discusses methods related to the latter task. The substructure drawn in blue in Figure 1 is similar to the substructure α, but they do not match perfectly. In the situation shown, the substructure α, which contributes to anti-influenza agents, is registered in a database. The database contains various substructures for which its medicinal effects, side effects and so on are known. If ribosylated favipiravir is given as the query and the substructure α is the output as the search result, we may be able to discover that ribosylated favipiravir is anti-influenza. Therefore, such a search engine for chemical compounds would be very useful and effective.

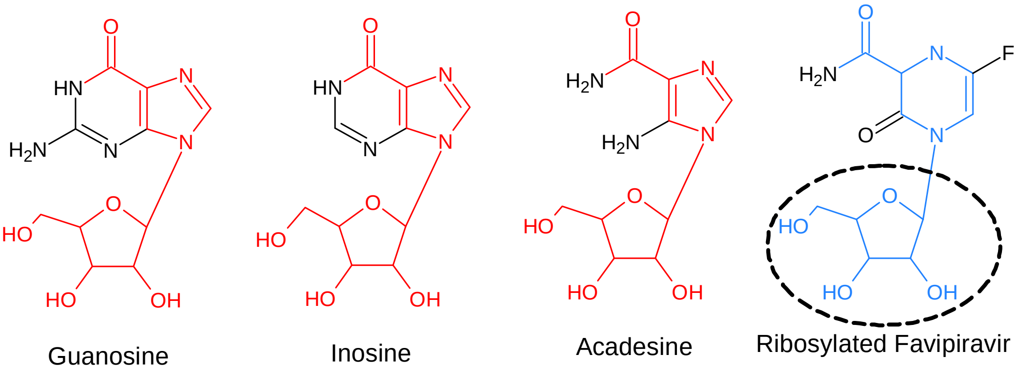


Figure 1. Similar Chemical Compounds.

A compound can be represented as a graph, in which the vertices and edges correspond to the atoms and chemical bonds, respectively. The aforementioned substructure and superstructure search correspond to subgraph search [2–14] and supergraph search [15–22], respectively, which are actively being studied by researchers. A graph is a highly versatile data structure for representing information in many fields. It can be used to represent human relationship networks, protein interaction networks, resource description frameworks (RDF), computer-aided designs (CAD), and computer vision images; moreover, subgraph and supergraph search are used for many applications in addition to searching for compounds. However, there is no method for searching for graphs that are similar to subgraphs of a query graph and no method applicable to the resemblance between substructures shown in Figure 1. This paper proposes the problem of similar supergraph search and also proposes a method to solve it.

In this paper, we measure the similarity between two graphs g and g' by the graph edit distance. The graph edit distance is the minimum number of edits required to transform g to another g', where an edit is either an insertion, deletion, or relabeling of a vertex or edge. The number of graphs obtained by editing the graph g θ times increases drastically, especially when the numbers of vertices and edges are large and the varieties of vertex and edge labels are wide. Such a drastic increase in edited graphs makes it difficult to find similar graphs. However, some real applications need to obtain a subset of graphs edited θ times. For example, editing the substructure in a broken ellipse of Figure 1 is not admissible, whereas editing the remainder of the blue substructure is admissible. By restricting an editable part in the structure, the number of graphs obtained by editing the graph g θ times is decreased. In our proposed method, users can search for their desired type of similar graphs in a database containing graphs and this search can be realized by simply rewriting only one function without the need to modify our proposed algorithm. Therefore, the method proposed in this paper is a general and flexible framework for searching for similar subgraphs of query graphs and is easily customizable for searching for the types of graphs desired by the user.

The remainder of the paper is organized as follows. Section 2 formalizes the novel problem of searching for graphs contained in a query graph in a database storing many graphs and Section 3 discusses related work. In Sections 4 and 5, we propose straightforward methods for a simpler problem in which the searched graphs are constrained. In Sections 6 and 7, we propose a novel method for solving the original problem by extending the above methods. In Section 8, we analyze the computational efficiency of the proposed method by using real-world datasets. Finally, we conclude the paper in Section 9.

2. Problem Definition

A labeled graph is expressed as $g = (V, E, \Sigma, \ell)$, where V is a set of vertices, $E \subseteq V \times V$ is a set of edges, Σ is a set of labels for the vertices and edges, and $\ell : V \cup E \rightarrow \Sigma$ is a function to assign the labels to the vertices and edges. We denote the set of all vertices in graph g as $V(g)$. A graph g is called a directed graph if the edges in g have directions;

otherwise, g is called an undirected graph. A sequence of edges from v_1 to v_2 is called a path and a graph (for a directed graph, all edges in the graph are replaced with undirected edges.) and a path that exists between any two vertices is called connected.

Given graphs $g = (V, E, \Sigma, \ell)$ and $g' = (V', E', \Sigma', \ell')$, $\forall v, v_1, v_2 \in V'$, when an injective function $\phi : V' \to V$ fulfills the following conditions, g' is called a subgraph of g, which is denoted as $g' \subseteq g$:

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

In addition, when $(\phi(v_1), \phi(v_2)) \in E$ iff $(v_1, v_2) \in E'$, we call g' an induced subgraph of g, which is denoted as $g' \subseteq_i g$. Although this paper focuses only on undirected and connected graphs, the methods proposed can easily be extended to directed or unconnected graphs.

The graph edit distance is the minimum number of edits required to transform a graph g to another graph g', where an edit is either an insertion, deletion, or relabeling of a vertex or edge. In this paper, we assume that the cost of each edit is 1. The number of edits between graphs $g = (V, E, \Sigma, \ell)$ and $g' = (V', E', \Sigma, \ell')$ is defined as follows:

$$dist(g, g', \varphi) = \sum_{i=1}^{|V'|} c_{i\varphi(i)} + \sum_{i=1}^{|V'|-1} \sum_{j=i+1}^{|V'|} c\Big((v_i, v_j) \to \Big(v_{\varphi(i)}, v_{\varphi(j)}\Big)\Big), \tag{1}$$

where we assume that $|V| \leq |V'|$ and V is extended to $V^+ = V \cup \{\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_{|V'|-|V|}\}$ [23]. In addition, when g is transformed to g', the mapping between V and V' is expressed as a bijective mapping $\varphi : V^+ \to V'$. c_{ij} is the cost to edit the vertex $v_i \in V^+$ to $v_j \in V'$ and $c((v_i, v_j) \to (v_{i'}, v_{j'}))$ is the cost to edit edge $(v_i, v_j) \in E$ to $(v_{i'}, v_{j'}) \in E'$. Editing $\varepsilon_i \in V^+$ to $v_j \in V'$ means performing a vertex insertion in g'. Therefore, the graph edit distance $ed(g, g')$ between the graphs g and g' is defined as follows:

$$ed(g, g') = \min_{\varphi \in \Phi(|V'|, |V'|)} dist(g, g', \varphi), \tag{2}$$

where $\Phi(\alpha, \beta)$ is a set of all possible β-permutations of the integers $1, 2, \ldots, \alpha$. The solution to Equation (2) is one of the quadratic assignment problems, which are known to be NP-complete.

Given a set of graphs $G = \{g_1, g_2, \ldots, g_n\}$, a query graph q, and a threshold θ as input, the problem tackled in this paper is to output a set of graphs.

$$S = \{g_i \in G \mid \exists q' \subseteq q \ s.t. \ ed(g_i, q') \leq \theta\}. \tag{3}$$

In Equation (3), note that q' is a subgraph of q but not an induced subgraph of q. When θ is 0 in Equation (3), the problem is equivalent to the original supergraph search problem. Therefore, our problem, represented by Equation (3), is an extension of the supergraph search problem.

3. Related Work

In Section 2, we defined the problem tackled in this paper. Recently, much attention has been focused on searching for graphs satisfying some conditions in a database consisting of multiple graphs. Table 1 summarizes graph search problems and the publications that tackle each problem. The subgraph search problem is to find graphs that contain the query graph and can be expressed as follows:

$$\{g_i \in G \mid q \subseteq g_i\},$$

whereas the supergraph search problem is to find graphs that the query graph contains and it can be expressed as follows.

$$\{g_i \in G \mid g_i \subseteq q\}.$$

These problems contain the subgraph isomorphism problem, which is known to be NP-complete. The similar graph search problem is to find graphs that are similar to the query graph, and can be expressed as

$$\{g_i \in G \mid ed(g_i, q) \leq \theta\}.$$

This problem contains the graph edit distance problem, which is known to be NP-complete. Various methods have been proposed to solve these problems. The problem tackled in this paper is to find graphs that are similar to subgraphs of the query graph and contains both the subgraph isomorphism problem and graph edit distance problem. This problem is a novel one, to the best of our knowledge. Since the subgraph isomorphism problem and graph edit distance problem are NP-complete, a naive method for solving either of the problems for a given query graph and for each graph in the database would be unrealistic.

Most of the methods proposed in the articles cited in Table 1 are based on filtering and verification and an index in each of the methods has a set of patterns P extracted from the database in advance. For example, for the supergraph search problem, in the filtering phase,

$$G' = G \setminus \bigcup_{p \in P} \{g_i \in G \mid p \not\subseteq q, g_i \subseteq q\}$$

is obtained. Subsequently, the verification phase checks whether $g \subseteq q$, for each graph $g \in G'$. Figure 2a shows four graphs in a database and two patterns contained in an index. The index has the information that $p_1 \subset g_1$, $p_2 \subset g_2$, $p_1 \subset g_4$, and $p_2 \subset g_4$. Given the query shown in Figure 2c, supergraph search methods first check whether $p_1 \subseteq q$ and $p_2 \subseteq q$ and then obtain $G' = \{g_1, g_3\}$ in the filtering phase. In the verification phase, they test whether $g_1 \subseteq q$ and $g_3 \subseteq q$. Since the number of vertices in p_2 is less than one for g_2, the computation time to check whether $p_2 \subseteq q$ becomes less than one for $g_2 \subseteq q$ and we do not need to check whether $g_2 \subseteq q$ in the filtering phase. The patterns in P are either paths, trees, cycles, or graphs extracted from databases. The patterns are extracted by the enumeration or mining method. The enumeration method enumerates all subgraphs in graphs in a database, where the subgraphs are restricted to either paths, trees, cycles, or graphs. In contrast, the mining method enumerates all frequent patterns [24], which are defined as

$$F(G, \sigma') = \{p \mid p \subseteq g_i, g_i \in G, \sigma(p, G) \geq \sigma'\}$$

from graphs in a database, where $\sigma(p, G)$ is defined as the number of graphs in G containing pattern p:

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

The extracted patterns are stored in a tree or lattice structure in the index of the graph database. Enumerating patterns and mining patterns from a database are time-consuming processes.

Table 1. Summary of graph search problems and references.

	Complete Matching	Similar Matching
graph search	$\{g_i \in G \mid q = g_i\}$	$\{g_i \in G \mid ed(g_i, q) \leq \theta\}$ [25–32]
supergraph search	$\{g_i \in G \mid g_i \subseteq q\}$ [15–22]	$\{g_i \in G \mid \exists q' \subseteq q \; s.t. \; ed(g_i, q') \leq \theta\}$ This paper (novel problem).
subgraph search	$\{g_i \in G \mid q \subseteq g_i\}$ [2–14]	$\{g_i \in G \mid \exists g \subseteq g_i \; s.t. \; ed(g, q) \leq \theta\}$ There are no papers to our best knowledge.

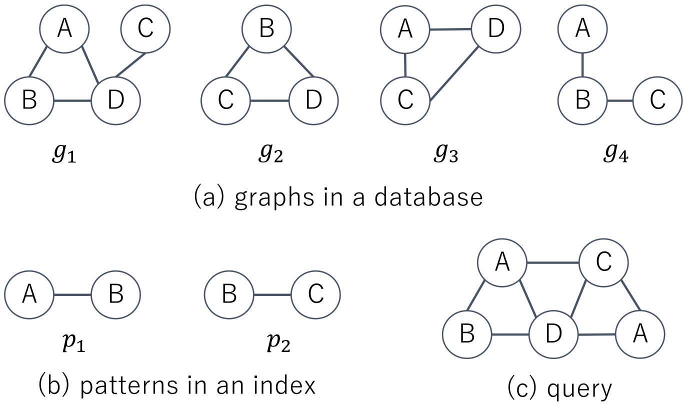


Figure 2. Example of supergraph search.

CodeTree [17] is a method for the supergraph search problem. It does not distinguish between filtering and verification phases; it filters out graphs that are not contained in a given query graph while verifying unfiltered graphs. In addition, CodeTree does not enumerate or mine patterns in a database. All graphs in a database are stored in the form of one of the graph codes, such as AcGM [33] code or DFS [34] code, in the index in CodeTree. The AcGM and DFS codes are based on adjacency matrices and adjacency lists, which are alternatives to graph representations. The algorithms for constructing an index for a database and searching graphs in the index are independent of the graph codes and, thus, the method can be integrated and extended with various graph codes for representing graphs in accordance with the intended characteristics of those codes. For this reason, the method proposed in this paper is based on CodeTree.

In this paper, we focus on the problems in which each database consists of multiple graphs. Other subgraph search problems are to search for a database which consists of "a single large graph". Since the number of graphs given as input is two, we need not distinguish the subgraph and supergraph searches. A typical problem of the graph search with a single large graph is to output all embeddings (or occurrences) $\{(\phi(v_1), \phi(v_2), \ldots, \phi(v_{|V(q)|}))\}$ in a large graph g for a query q consisting of vertices $(v_1, v_2, \ldots, v_{|V(q)|})$ when the query q is a subgraph of the large graph g. This problem is applicable to RDFs (Resource Description Frameworks), where each subject or object and predicate corresponds to a vertex and edge in a graph, respectively; PPI (Protein-Protein Interaction) networks, where each protein and interaction correspond to a vertex and edge, respectively; and intrusion alert networks where each computer and possible attack such as Denial-of-Service and TCP Service Sweep correspond a vertex and edge, respectively. Some of the methods for solving these kinds of problems use not only exact matching [35–37] but also approximate matching [38–41] with various similarity measures.

The most representative metric for measuring similarity between two graphs is the graph edit distance. Most of methods for graph search problems use this graph edit distance to measure similarities among graphs. Other most representative measures are to use maximum common subgraphs or graph kernels [42,43] between two graphs. There are two types of problems called maximum common induced subgraph problem [44] and maximum common edge subgraph problem [45] and the problems for finding the maximum common subgraphs in two graphs is NP-complete. In contrast, graph kernels have been proposed to classify graphs in the machine learning domain. The random walk graph kernel [46] returns a high similarity of graphs if random walks on the graphs produce similar sequences of vertex and edge labels. The Weisfeiler–Lehman graph kernel [47] uses the Weisfeiler–Lehman test, which was proposed to solve the graph isomorphism problem. The kernel iteratively relabels each vertex v by concatenating a label of v and labels of v's adjacent vertices and returns high similarity for two relabeled graphs if the graphs have many common labels.

4. Straightforward Method for Constrained Problem

In Section 2, we defined the problem tackled in this paper. Here, for the sake of simplicity, we first provide methods for outputting the following:

$$\{g_i \in G \mid \exists q' \subseteq_i q \text{ s.t. } ed(g_i, q') \leq \theta, q' \text{is connected}, |V(g_i)| \leq |V(q')|\}, \tag{4}$$

by constraining the original problem. In the subsequent sections, we explain how the method can be extended to solve the original problem.

Definition 1 (AcGM code [33])**.** *Given a graph $g = (V, E, \Sigma, \ell)$, we assign vertex identifiers $u_1, u_2, \ldots, u_{|V|}$ to vertices in g and assume that a subgraph of g induced by vertices $u_1, u_2, \ldots, u_i$ $(1 \leq i \leq |V|)$ is connected. g is represented by an adjacency matrix of dimension $|V| \times |V|$ for which its elements $x_{i,j}$ are $\ell((u_i, u_j))$ if $(u_i, u_j) \in E$ or 0 otherwise. The AcGM code of graph g is defined by concatenating the elements in the upper-right part of the adjacency matrix, as follows:*

$$code(g, \langle u_1, u_2, \ldots, u_{|V|} \rangle) = s_1 s_2 \cdots s_{|V|}, \tag{5}$$

where each code fragment s_i is defined as follows:

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

$s_1 s_2 \cdots s_i$ is called a prefix of Equation (5). In addition, $c_2 \subseteq c_1$ means that a prefix of an AcGM code c_1 is equal to another AcGM code c_2 and a graph expressed by AcGM code c is denoted as $g(c)$.

Definition 2 (Linear ordering between AcGM codes)**.** *When two AcGM codes $\alpha = a_1 a_2 \cdots a_k$ and $\beta = b_1 b_2 \cdots b_h$ fulfill either of the following conditions, we say that $\beta \preceq \alpha$:*

- $\exists t$ *s.t.* $1 \leq t \leq min(k, h)$, $a_q = b_q$ *for* $q < t$ *and* $b_t \prec_e a_t$;
- $\beta \subseteq \alpha$;

where $\prec_e$ is a linear ordering between code fragments [34].

Example 1. *Given the graph with four vertices shown in Figure 3, we have $4! = 24$ possible permutations of the four vertices. Among the permutations, according to Definition 1, $code(g, \langle v_2, v_3, v_4, v_1 \rangle)$ does not generate an AcGM code because a subgraph induced by $\langle v_2, v_3 \rangle$ is not connected. We can generate 12 AcGM codes from the graph and we show three AcGM codes as follows. Symbols on the left side of the following adjacent matrices are vertex IDs and the letters on the upper side of the matrices are vertex labels. In addition, letters in the matrices are edge labels and zeros in the matrices and they indicate that there are no edges.*

$$\begin{array}{ccccc} & A{\downarrow} & B & C & B \\ v_1 & 0 & a{\downarrow} & b & a \\ v_2 & a & 0 & 0{\downarrow} & 0 \\ v_3 & b & 0 & 0 & 0{\downarrow} \\ v_4 & a & 0 & 0 & 0 \end{array}, \quad \begin{array}{ccccc} & B{\downarrow} & A & C & B \\ v_2 & 0 & a{\downarrow} & 0 & 0 \\ v_1 & a & 0 & b{\downarrow} & a \\ v_3 & 0 & b & 0 & 0{\downarrow} \\ v_4 & 0 & a & 0 & 0 \end{array}, \quad \begin{array}{ccccc} & B{\downarrow} & A & B & C \\ v_4 & 0 & a{\downarrow} & 0 & 0 \\ v_1 & a & 0 & a{\downarrow} & b \\ v_2 & 0 & a & 0 & 0{\downarrow} \\ v_3 & 0 & b & 0 & 0 \end{array}$$

By concatenating the letters and zeros on sides of arrows in the matrices, three AcGM codes are generated as follows.

$$\begin{aligned} \alpha &= code(g, \langle v_1, v_2, v_3, v_4 \rangle) = A\ Ba\ Cb0\ Ba00, \\ \beta &= code(g, \langle v_2, v_1, v_3, v_4 \rangle) = B\ Aa\ C0b\ B0a0, \text{and} \\ \gamma &= code(g, \langle v_4, v_1, v_2, v_3 \rangle) = B\ Aa\ B0a\ C0b0. \end{aligned}$$

The order between the codes is $\alpha \prec \gamma \prec \beta$.

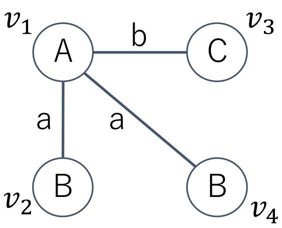


Figure 3. Example of a graph.

Lemma 1. *If two AcGM codes are the same, the graphs represented by the codes are isomorphic.*

Proof. If two adjacency matrices are the same, the graphs represented by the matrices are isomorphic. Since each AcGM code of a graph g corresponds one-to-one with an adjacency matrix of g, if two AcGM codes are the same, the graphs represented by the codes are isomorphic. □

In contrast, if two graphs are isomorphic, their AcGM codes are not necessarily the same. This means that there are multiple AcGM codes of g that fulfill Definition 1. Thus, we denote the set of all AcGM codes of graph g as $\Omega(g)$.

Lemma 2. *When an AcGM code c is a prefix of another code c', $g(c)$ is connected and is an induced subgraph of $g(c')$.*

Proof. Given an AcGM code c of a graph g, let its corresponding adjacency matrix be A. All submatrices A' of A obtained by iteratively deleting the last row and column of A represent induced and connected subgraphs of g. According to Definition 5, AcGM codes for the matrices A' must be prefices of the AcGM code for A. □

From the above discussion, one straightforward method for outputting the subgraphs defined by Equation (4) is to compute the graph edit distance between a graph $g_i \in G$ and a graph $g(c')$, represented by a prefix $c' \subseteq c$ for $c \in \Omega(q)$. However, computing the edit distance between g_i and $g(c')$ by Equation (2) is not feasible. Therefore, we discuss a method for computing the edit distance by using the codes of g_i and $g(c')$.

Lemma 3. *Given the following AcGM codes:*

$$c = \ell(u_1)\ell(u_2)x_{1,2}\cdots\ell(u_a)x_{1,a}\cdots x_{a-1,a} \tag{6}$$
$$c' = \ell(u'_1)\ell(u'_2)x'_{1,2}\cdots\ell(u'_b)x'_{1,b}\cdots x'_{b-1,b}, \tag{7}$$

where a and b are the numbers of vertices in $g(c)$ and $g(c')$, respectively, and the i-th vertex in $g(c)$ maps to the i-th vertex in $g(c')$, the number of edits $ed(c,c')$ to transform $g(c)$ to $g(c')$ is the following:

$$\begin{aligned} ed(c,c') &= \sum_{1\le i\le \min\{a,b\}} \delta'(\ell(u_i),\ell(u'_i)) + \sum_{1<j\le \min\{a,b\}} \sum_{1\le i<j} \delta'(x_{i,j},x'_{i,j}) \\ &\quad + |a-b| + \begin{cases} \sum_{b<j\le a}\sum_{1\le i<j}\delta'(x_{i,j},0) & (b<a) \\ 0 & (a=b) \\ \sum_{a<j\le b}\sum_{1\le i<j}\delta'(0,x'_{i,j}) & (a<b), \end{cases} \end{aligned} \tag{8}$$

where δ is the Kronecker delta and $\delta'(\alpha,\beta) = 1-\delta(\alpha,\beta)$. The complexity of solving Equation (8) is $O((\max\{a,b\})^2)$. The $ed(c,c')$ function is symmetric: $ed(c,c') = ed(c',c)$.

Proof. The first term of Equation (8) is the sum of the numbers of edits that occur because of differences between the i-th ($1 \le i \le \min\{a,b\}$) vertices in $g(c)$ and $g(c')$ and its second term is the sum of the numbers of edits that occur because of differences between edges

(i,j) $(1 \leq i < j \leq \min\{a,b\})$ in $g(c)$ and $g(c')$. Its third term is the sum of the numbers of edits to insert vertices in $g(c)$ or $g(c')$ and its fourth term is the sum of the numbers of edits to insert edges in $g(c)$ or $g(c')$. □

Since Equation (8) defines the number of edits between two graphs under the known mapping such that $\varphi(i) = i$ for all i $(1 \leq i \leq \max\{a,b\})$, it corresponds to Equation (1). Therefore, Equation (2) is formalized by using Equation (8) as the following lemmas.

Lemma 4. *Given the two AcGM codes c and c' defined in Equations (6) and (7), we have the following.*

$$ed(g(c), g(c')) \leq ed(c, c'). \tag{9}$$

Proof. According to Equation (2), $ed(g(c), g(c')) \leq ed(c, c')$ is obvious. □

Lemma 5. *Given an arbitrary AcGM code c' of a graph g', the graph edit distance between g and g' is the following.*

$$ed(g, g') \leq \min_{c \in \Omega(g)} ed(c, c'). \tag{10}$$

Proof. Let $\Omega^+(g)$ be codes generated from all possible permutations of vertices in g. So $\Omega^+(g) = \{code(g, \langle u_1, u_2, \ldots, u_{|V(g)|}\rangle) \mid \langle u_1, u_2, \ldots, u_{|V(g)|}\rangle \in \Phi(|V(g)|, (|V(g)|)\} \supseteq \Omega(g)$. Some codes c^+ in $\Omega^+(g)$ are not AcGM codes because a prefix of the permutation from which c^+ is generated may not induce a connected subgraph of g. Since $\Omega^+(g)$ are codes generated from all possible permutations of vertices in g, we have $ed(g, g') = \min_{c \in \Omega^+(g)} ed(c, c')$. In addition, because $\Omega(g) \subseteq \Omega^+(g)$, we have $\min_{c \in \Omega^+(g)} ed(c, c') \leq \min_{c \in \Omega(g)} ed(c, c')$. Therefore,

$$ed(g, g') = \min_{c \in \Omega^+(g)} ed(c, c') \leq \min_{c \in \Omega(g)} ed(c, c')$$

is satisfied. □

Example 2. *As shown in Figure 4, given two graphs g and q for which the edit distance is 3, let an arbitrary AcGM code of g be AAb. $\Omega(q)$ is $\{AAaB0a, AAaBa0, ABaAa0, BAaA0a\}$. When the code AAb is compared with $AAaB0a \in \Omega(q)$, we need three edits which are indicated as × in Figure 4. By comparing AAb and all elements in $\Omega(q)$, we obtain the graph edit distance between q and g according to Lemma 10.*

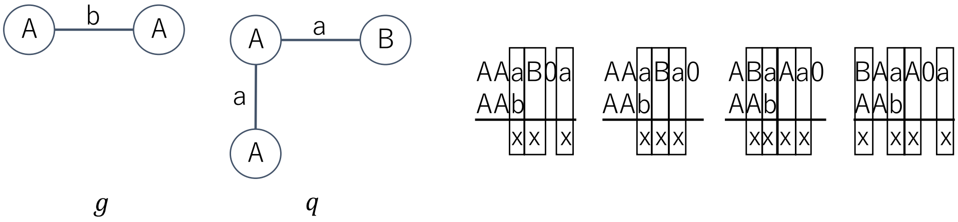


Figure 4. Computation of Graph Edit Distance using AcGM codes.

From Lemma 5, we have the following corollary.

Collorary 1. *If $ed(c, c') \leq \theta$ for two AcGM codes c and c', then $ed(g(c), g(c')) \leq \theta$.*

Algorithm 1 shows the pseudocode for solving the problem defined by Equation (4). In Line 3, an arbitrary AcGM code from $\Omega(g_i)$ is generated for graph g_i in database G. After one of all possible AcGM codes for q is obtained in Line 4, a prefix c', which consists of j fragments c_i, is generated in Line 6. Since c' is a prefix of c, $g(c')$ must be a connected and is an induced subgraph of q. Subsequently, g_i is added to the set of output graphs S if $ed(c_i, c') \leq \theta$, which indicates that g_i is a solution in the set defined by Equation (4) according to Corollary 1. Once g_i is added to S, any procedures for g_i are stopped and procedures for the next graph g_{i+1} are started. These steps are repeated for all of the graphs in G. The time complexity of Algorithm 1 is $O(|\Omega(q)| \sum_{i=1}^{|G|} |V(g_i)|^3)$ and the algorithm has some drawbacks, as follows.

(1) Although the algorithm produces $|V(g_i)|$ prefixes in Line 6 for each AcGM code c in $\Omega(q)$ and computes Equation (8) $|V(g_i)|$ times in Line 7 for the prefixes, some of the repeated calculations of Equation (8) are redundant because two prefixes c' and c'' produced from c satisfy $c' \subset c''$. Using the result of computing $ed(c_i, c')$ to compute $ed(c_i, c'')$ would render Algorithm 1 efficient.

(2) Let $G_c = \{code(g_i) \mid g_i \in G\}$ be the set of codes produced in Line 3. AcGM codes $code(g_i)$ and $code(g_j)$, for which their prefixes are the same, are included in G_c. The repeated calculations of Equation (8) between these AcGM codes and c' are redundant. If the common prefix of $code(g_i)$ and $code(g_j)$ is c_s, using the result of computing $ed(c_s, c')$ to compute $ed(code(g_i), c')$ and $ed(code(g_j), c')$ would make Algorithm 1 efficient.

Algorithm 1: Straightforward Algorithm for Searching (4)

```
   h!
   Data: $G = \{g_1, g_2, \ldots, g_n\}, q, \theta$
   Result: $\{g_i \in G \mid \exists q' \subseteq_i q \ s.t.\ ed(g_i, q') \leq \theta, q'$is connected, $|V(g_i)| \leq |V(q')|\}$
1  $S \leftarrow \emptyset$; //a variable for storing solutions
2  for $g_i \in G$ do
3      $c_i \leftarrow code(g_i)$; //an arbitrary AcGM code of $g_i$
4      for $c \in \Omega(q)$ do
5          for $j \in [1, |V(g_i)|]$ do
6              $c' \leftarrow pre(c, j)$;
7              if $ed(c_i, c') \leq \theta$ then
8                  $S \leftarrow S \cup \{g_i\}$;
9                  break;
10         if $g_i \in S$ then
11             break;
12 return $S$;
```

5. Method for Traversing Prefix Tree for Constrained Problem

In order to overcome drawback (1), described in the previous section, we introduce the following lemma.

Lemma 6. *Given two AcGM codes c and c', we have the following:*

$$ed(pre(c,i), pre(c',i)) \leq ed(pre(c,j), pre(c',j)) \text{ for } i < j. \tag{11}$$

In the case that i is greater than the number of fragments in c, we assume that $pre(c,i)$ is equivalent to c itself.

Proof. $ed(pre(c,i+1), pre(c',i+1))$ is the sum of $ed(pre(c,i), pre(c',i))$ and the number of edits between the $i+1$-th fragments of $pre(c,i+1)$ and $pre(c',i+1))$. Since the number of edits is non-negative, $ed(pre(c,i), pre(c',i))$ increases monotonically when i is increased. Therefore, for $i < j$, we have $ed(pre(c,i), pre(c',i)) \leq ed(pre(c,j), pre(c',j))$. □

Collorary 2. *According to Lemma 6, if $ed(pre(c,i), pre(c',i)) > \theta$, then $ed(c,c') > \theta$.*

As a consequence of Lemma 6 and Corollary 11, when $ed(pre(c,i), pre(c',i)) > \theta$, we can stop the computation in Line 7 of Algorithm 1.

Lemma 7. *Given the two following fragments:*

$$s = \ell(u_i)x_{1,i} \cdots x_{i-1,i} \tag{12}$$
$$s' = \ell(u_i')x_{1,i}' \cdots x_{i-1,i}', \tag{13}$$

for which their lengths are equal, the number of edits $ed(s,s')$ to equalize the fragments is as follows:

$$ed(s,s') = \begin{cases} 1 + \sum\limits_{1 \le j < i} \delta'(0, x_{j,i}') & (s = null) \\ 1 + \sum\limits_{1 \le j < i} \delta'(x_{j,i}, 0) & (s' = null) \\ \delta'(\ell(u_i), \ell(u_i')) + \sum\limits_{1 \le j < i} \delta'(x_{j,i}, x_{j,i}') & (otherwise). \end{cases} \tag{14}$$

Since $ed(s,s')$ is a symmetric function, $ed(s,s') = ed(s',s)$.

Proof. Please see the proof for Lemma 3. □

Algorithm 2 shows the pseudocode for solving the problem defined by (4), which overcomes the aforementioned drawback (1). After obtaining the j-th fragments of AcGM codes c_i and c in Line 7 of the algorithm, the number of edits between the fragments is calculated according to Lemma 7. The number of edits is then added to the accumulated edit count ae. According to Corollary 2, the algorithm stops the calculation of the number of edits between c_i and one of the AcGM codes of q in Line 8. However, because g_i is a member of the set defined in Equation (4) if the edited distance between c_i and at least one AcGM code in $\Omega(q)$ is less than θ, these steps are repeated for all AcGM codes of q. The time complexity of Algorithm 2 is $O(|\Omega(q)| \sum_{i=1} |G||V(g_i)|^2)$.

Algorithm 2: Straightforward Algorithm 2 for Searching (4)

```
   Data: $G = \{g_1, g_2, \ldots, g_n\}$, $q$, and $\theta$
   Result: $\{g_i \in G \mid \exists q' \subseteq_i q \ s.t.\ ed(g_i, q') \le \theta, q'$is connected, $|V(g_i)| \le |V(q')|\}$
1  $S \leftarrow \emptyset$; //a variable for storing solutions
2  for $g_i \in G$ do
3      $c_i \leftarrow code(g_i)$;
4      for $c \in \Omega(q)$ do
5          $ae \leftarrow 0$; //$ae$ is accumulated edit count
6          for $j \in [1, |V(g_i)|]$ do
7              $ae \leftarrow ae + ed(frag(c_i, j), frag(c, j))$;
8              if $ae > \theta$ then
9                  break;
10             if $j = |V(g_i)|$ then
11                 $S \leftarrow S \cup \{g_i\}$;
12         if $g_i \in S$ then
13             break;
14 return $S$;
```

In order to overcome drawback (2), described in the previous section, we use a prefix tree for the AcGM codes for G.

Definition 3 (Code Tree [17] (The detailed algorithm for constructing a code tree from a graph database G is given in [17].))**.** *A code tree T is defined as the triplet $(\top, N, B)$, where*

$\top \in N$ is the root node of the tree, N is a set of the nodes of the tree, and B is a set of the branches of the tree. In addition, each node is associated with a code fragment and a set of graph identifiers. Let $s(n)$ be the concatenation of the fragments associated with nodes on the path from the root to node n. When one of the AcGM codes of $g_i \in G$ is the same as $s(n)$, the set of graph identifiers for node n contains i, which is the identifier of graph $g_i \in G$.

The code fragment and graph identifiers associated with node n are denoted as $fr(n)$ and $ID(n)$, respectively. For the root node, $fr(\top) = null$ and $ID(\top) = \emptyset$. We present an example of a code tree below.

Example 3. *A graph database $G = \{g_1, g_2, g_3, g_4\}$ consists of four graphs, as shown in Figure 5. When one of the AcGM codes for each graph in G is generated, the AcGM codes for the four graphs are as follows.*

$$\begin{aligned} code(g_1, \langle v_1, v_2, v_3 \rangle) &= ABaC0b, \\ code(g_2, \langle v_1, v_2, v_3 \rangle) &= ABaD0b, \\ code(g_3, \langle v_1, v_2 \rangle) &= BBa, and \\ code(g_4, \langle v_1, v_2, v_3 \rangle) &= BBaC0b. \end{aligned}$$

From these AcGM codes, a code tree is constructed, as shown in the right part of Figure 5. Node n_6 of the tree is associated with the code fragment $C0b$ and the set of graph identifiers $ID(n) = \{1\}$. The concatenation of fragments associated with nodes on the path from the root to node n_6 is $ABaC0b$, which represents graph g_1 in G.

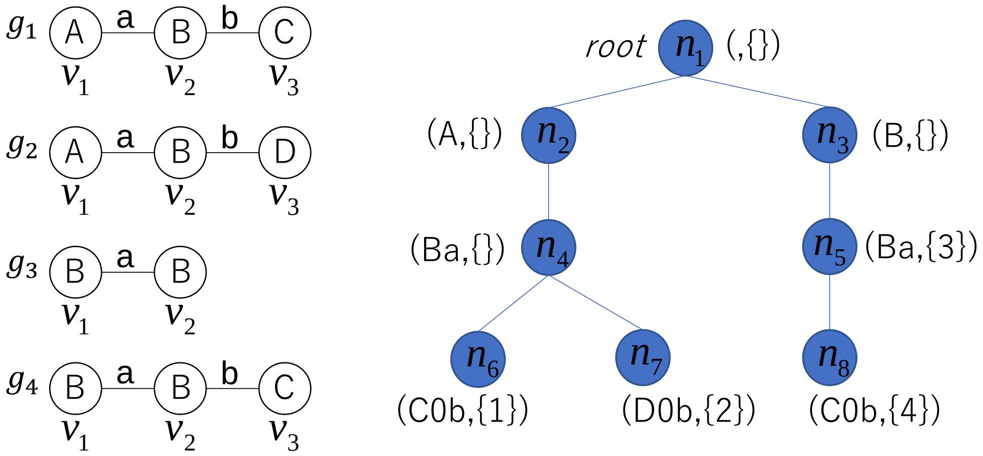


Figure 5. Code tree.

Algorithm 3 shows the pseudocode for searching for members of the set defined in Equation (4) by traversing the code tree. At node n, Algorithm 3 adds vertex w of q to $\langle w_1, w_2, \ldots, w_h \rangle$, from which the fragment s is generated, with the addition of the prefix $code(q, \langle w_1, w_2, \ldots, w_h \rangle)$ of an AcGM code of q. The algorithm then recursively traverses the children of n that have fragments similar to fragment s. In contrast to Algorithm 2, which iteratively checks whether each individual graph in the graph database is a solution, Algorithm 3 checks whether multiple graphs in the database are simultaneously solutions. This is because each node in the code tree is associated with the common prefix of the AcGM codes of multiple graphs in the database. This simultaneous checking renders our search very efficient. The worst case time complexity of Algorithm 3 is the same as the time complexity of Algorithm 2. When AcGM codes of graphs in the database do not have common prefixes as one another, the time complexity of Algorithm 3 becomes worst. However, the codes usually have common prefixes as one another and the computation time that this algorithm needs is proportional to the number of nodes that Algorithm 3 traverses. In addition, at each node, Algorithm 3 needs $O(h|V(q)||N|)$.

Algorithm 3: Code Tree Search for Finding Solutions to Equation (4)

```
   Data: the set of graphs $S$, query $q$, threshold $\theta$, current node $n$, $\langle w_1, w_2, \ldots, w_h \rangle$,
         and accumulated edit count $ae$
   Result: $\{g_i \in G \mid \exists q' \subseteq_i q \; s.t. \; ed(g_i, q') \leq \theta, q'$is connected, $|V(g_i)| \leq |V(q')|\}$
1  $S \leftarrow S \cup \bigcup_{i \in ID(n)} \{g_i\}$;
2  $N \leftarrow children(n)$;
3  $C \leftarrow \{(w, s) \mid s_1 s_2 \cdots s_h s = code(q, \langle w_1, w_2, \ldots, w_h, w \rangle) \subseteq c, c \in \Omega(q)\}$;
4  for $(m, (w, s)) \in N \times C$ do
5      $e \leftarrow ed(fr(m), s)$;
6      if $ae + e \leq \theta$ then
7          $\omega \leftarrow \langle w_1, w_2, \ldots, w_h, w \rangle$;
8          $S \leftarrow search(S, q, \theta, m, \omega, ae + e)$;
9  return $S$;
```

6. Method for Traversing Prefix Tree for Original Problem

At node n, for which its depth is h in the code tree, Algorithm 3 retains the accumulated edit count ae for two concatenations c and c' of fragments of length h. c and c' have the following restrictions:

- c is a concatenation of fragments associated with nodes on the path from the root of the code tree to n and is an AcGM code of a connected and induced graph that is a common subgraph of multiple graphs in the graph database;
- c' is a prefix of one of the AcGM codes in $\Omega(q)$. According to the definition of the AcGM code, $g(c')$ must be connected and is an induced subgraph of q.

In order to solve the original problem for searching for solutions to Equation (3), we relax the restrictions as follows. The restriction that $g(c')$ is connected is a consequence of the fact that $code(q, \langle w_1, w_2, \ldots, w_h \rangle)$ is a prefix of the AcGM codes $\Omega(q)$ of q. Therefore, in Line 4 of Algorithm 3., we relax this restriction by allowing $\langle w_1, w_2, \ldots, w_h \rangle$ to be all possible h-permutations of $vertices\{v_1, v_2, \ldots, v_{|V(q)|}\}$ in q. Codes $code(q, \langle w_1, w_2, \ldots, w_h \rangle)$ by some of the permutations correspond to unconnected graphs.

Next, we relax the restriction that $g(c')$ is an induced subgraph of q. Algorithm 1 produces prefixes of the AcGM codes of q to obtain connected and induced subgraphs of q. Suppose that we would like to obtain, in addition to the connected and induced subgraphs, all possible unconnected and non-induced subgraphs of q. In this case, we may need, in addition to the prefixes of the AcGM codes of q, all possible codes generated by replacing some elements $x_{i,j}$ in the prefixes by 0. This requires us to solve both the permutation problem among the vertices of q and the combination problem among the edges of q. However, we do not need to solve the latter problem for the following reasons. We consider a graph q, its induced subgraph q', and two graphs g_1 and g_2, shown in Figure 6. We refer to two vertices in q' as v_i and v_j.

- $ed(q', g_1) = 2$. Given a graph q'' obtained by removing an edge from q', we have $ed(q', g_1) < ed(q'', g_1) = 3$. In the problem of finding solutions to Equation (3), we check whether there exists a subgraph of q such that $ed(q', g_i) \leq \theta$ and this subgraph need not be an induced subgraph of q. Therefore, in the case that there is an edge between two vertices v_i and v_j in q' and there is also an edge in g_i between the two corresponding vertices, we do not need to consider the graph q'' obtained by removing an edge from q'. That is, we do not need to replace any elements $x_{i,j}$ in prefixes of the AcGM codes of q' by 0.
- $ed(q', g_2) = 1$. For the above q'', we have $ed(q', g_2) > ed(q'', g_2) = 0$. Therefore, in the case that there is an edge between two vertices v_i and v_j in q' and there is no edge in g_i between the two corresponding vertices, we do not need to consider the graph that does not have an edge corresponding to edge (v_i, v_j) in q'.

- In all other cases, there is no edge between v_i and v_j in q'. Since $x_{i,j} = 0$, we do not need to replace $x_{i,j}$ by 0.

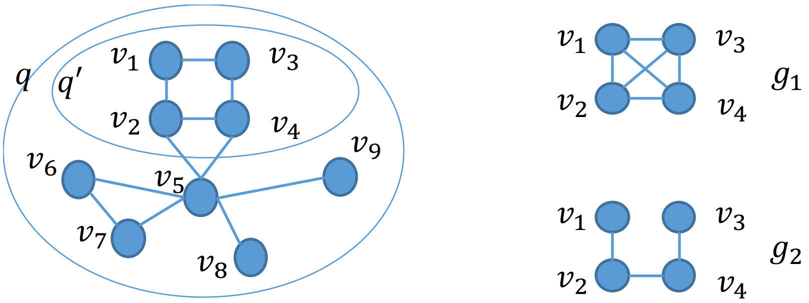


Figure 6. Inclusion of graphs and similarity.

From the above discussion, we rewrite Lemma 7 as follows.

Lemma 8. *$ed(s, s')$*

$$= \begin{cases} 1 + \sum_{1 \le j < i} \delta'(x_{j,i}, 0) & (s' = null) \\ 1 & (s = null) \\ \delta'(\ell(v_i), \ell(v'_i)) + \sum_{\substack{1 \le j < i \\ x_{i,j} \neq 0}} \delta'(x_{j,i}, x'_{j,i}) & (s \neq null \wedge s' \neq null). \end{cases} \tag{15}$$

This function is asymmetric.

Since we consider all possible subgraphs of q by the function defined in Equation (8), we need to produce only the prefixes underlined above.

Algorithm 4 shows the pseudocode for searching for members of the set defined in Equation (3) by traversing the code tree. The part from Lines 4–9 searches $\{g_i \in G \mid q' \subseteq q \text{ s.t. } ed(q', g_i) \le \theta, |V(g_i)| \le |V(q')|\}$, whereas the part from Lines 10–13 searches $\{g_i \in G \mid q' \subseteq q \text{ s.t. } ed(q', g_i) \le \theta, |V(g_i)| > |V(q')|\}$. The other difference between Algorithm 3 and Algorithm 4 is Line 4, which produces codes from one of all possible h-permutations of vertices of the query graph q. The codes produced are not limited to AcGM codes, but are AGM codes [24]. In Lines 7 and 11, Equation (15) is used to calculate the similarity between two fragments. Since the codes produced in Line 4 are limited to prefixes of the AGM codes of q, we do not need to solve the combination problem among the edges of q, which enables our proposed method to be very efficient. The time complexity of Algorithm 4 is the same as one of Algorithm 3; nevertheless, the problem that Algorithm 3 solves is a constrained problem of the problem that Algorithm 4 solves.

Algorithm 4: Code Tree Search for Finding Solutions to Equation (3)

```
Data: The set of graphs $S$, query $q$, threshold $\theta$, current node $n$, $\langle w_1, w_2, \ldots, w_h \rangle$,
      and accumulated edit count $ae$.
Result: $\{g_i \in G \mid q' \subseteq q \; s.t. \; ed(q', g_i) \leq \theta\}$
$S \leftarrow S \cup \bigcup_{i \in ID(n)} \{g_i\}$;
$N \leftarrow children(n)$;
if $\omega \neq null \wedge h < |V(q)|$ then
    $C \leftarrow \{(w, s) \mid s_1 s_2 \cdots s_h s = code(q, \langle w_1, w_2, \ldots, w_h, w \rangle), \langle w_1, w_2, \ldots, w_h, w \rangle \in$
      $\Phi(|V(q)|, h+1)\}$;
    for $(m, (w, s)) \in N \times C$ do
        $\omega \leftarrow \langle w_1, w_2, \ldots, w_h, w \rangle$;
        $e \leftarrow ed(fr(m), s)$;
        if $ae + e \leq \theta$ then
            $S \leftarrow search(S, q, \theta, m, \omega, ae + e)$;
else
    for $m \in N$ do
        $e \leftarrow ed(fr(m), null)$;
        if $ae + e \leq \theta$ then
            $S \leftarrow search(S, q, \theta, m, null, ae + e)$;
return $S$;
```

7. Customization of Solutions

In the previous section, we proposed a method for searching for the set of graphs S defined by Equation (3). The number of graphs reachable by editing graph q' θ times in Equation (3) increases exponentially as θ is increased. Therefore, for a large θ, the computation time required to search for the graphs in S becomes excessively long. However, some real applications need to obtain the following sets of constrained graphs rather than obtaining all the graphs in S.

⟨1⟩ Editing graphs is constrained: for example, the relabeling of vertices or edges is admissible in $ed(q', g_i)$ of Equation (3), whereas insertions and deletions are not admissible. That is, when converting a labeled graph g to an unlabeled graph $un(g)$ by removing label information from vertices and edges in g, $un(g_i)$ and $un(q')$ in Equation (3) are isomorphic.

⟨2⟩ Editing graphs is constrained: for example, editing some specific vertices and edges in a query graph q is admissible, whereas editing other vertices and edges is not admissible. For example, in the substructure drawn with blue lines and labels in Figure 1, editing its ring structure is admissible, whereas editing the remainder of the blue substructure is not admissible.

The problem for the above constraint ⟨1⟩ is formalized as follows.

$$\{g_i \in G \mid \exists q' \subseteq q \; s.t. \; ed(g_i, q') \leq \theta \wedge un(g_i) = un(q')\}. \tag{16}$$

In order to solve this problem, Equation (15) is rewritten as follows.

$$ed(s, s') = \begin{cases} \delta'(\ell(v_i), \ell(v'_i)) + \displaystyle\sum_{\substack{1 \leq j < i \\ \wedge x_{i,j} \neq 0 \\ \wedge x'_{i,j} \neq 0}} \delta'(x_{j,i}, x'_{j,i}) + \displaystyle\sum_{\substack{1 \leq j < i \\ \wedge (x_{i,j} = 0 \\ \vee x'_{i,j} = 0)}} \infty & (s \neq null \wedge s' \neq null) \\ \infty & (otherwise). \end{cases} \tag{17}$$

For the case of $s \neq null \wedge s' \neq null$, the first and second terms in Equation (17) are the numbers of edits that occur by relabeling the i-th vertices and edges (i, j) $(1 \leq j < i)$, respectively, in $g \in G$ and q'. The third term of Equation (17) is the sum of the numbers of edits that occur by inserting or deleting edges. For this case, in containing such insertion or deletion operations, the $ed(s, s')$ function returns values larger than θ and Algorithm 4 (in Line 7) backtracks the code tree.

For the case of the above constraint $\langle 2 \rangle$, let some specific vertices in query graph q be $V_q \subseteq V(q)$. When Algorithm 4 visits node n in the code tree, it generates various codes by $\langle w_1, w_2, \ldots, w_h, w \rangle$, in Line 4. Some vertices in $\langle w_1, w_2, \ldots, w_h, w \rangle$ are included in V_q, but others are not. In this case, Equation (15) is rewritten as follows.

$$ed(s, s') = \begin{cases} \delta'(\ell(v_i), \ell(v'_i)) + \sum\limits_{\substack{w_j \in V_q \\ \wedge 1 \le j < i \\ \wedge x_{i,j} \neq 0}} \delta'(x_{j,i}, x'_{j,i}) + \sum\limits_{\substack{w_j \notin V_q \\ \wedge 1 \le j < i \\ \wedge x_{i,j} \neq 0}} \infty & (s, s' \neq null \wedge w_i \in V_q) \\ \infty & (otherwise). \end{cases} \quad (18)$$

Thus, for the case of $s \neq null \wedge s' \neq null \wedge w_i \in V_q$, the first and second terms of Equation (18) are the numbers of edits that occur by relabeling the i-th vertices and edges (i, j) $(1 \le j < i)$, respectively, in $g \in G$ and q', where w_i and w_j are elements in V_q. The third term of Equation (18) is the sum of the numbers of edits that occur by editing the other vertices and edges.

In order to summarize the above discussion, the advantages of our proposed method are the following:

- Users can search for their desired type of similar graphs in a database containing graphs;
- This search can be realized by simply rewriting Equation (15), without the need to modify Algorithm 4.

Therefore, the method proposed in this paper is a general and flexible framework for searching for similar subgraphs of query graphs and is easily customizable for searching for the types of graphs desired by the user.

8. Experimental Evaluation

For our experimental evaluation, we used the three real-world datasets used in [18] which were downloadable from https://github.com/SNUCSE-CTA/IDAR on 1 March 2021. The datasets consisting of chemical compounds are called AIDS, NCI, and PubChem. Each atom, chemical bond, atom type, and bond type in a chemical compound corresponds to a vertex, edge, vertex label, and edge label, respectively. Sets of queries and databases containing graphs were generated from each of the datasets. One hundred query graphs, each containing more than 100 vertices, were selected from each of the datasets. Each generated database contained between 10,000 and 100,000 graphs and each of the graphs was generated by one of two different methods [18]. In the first method, each graph in a database consisted of vertices and edges on a random walk on a graph randomly selected from one of the datasets. This type of database is called *rand*. In the second method, each graph in a database was a frequent subgraph with a minimum support value of 0.1% [24] that is enumerated from one of the datasets. This type of database is called *freq*. In the experiments, vertices corresponding to hydrogen were removed, which is similar to the experiments in [2,14,17,30]. Table 2 summaries graphs in databses and queries in our experiments.

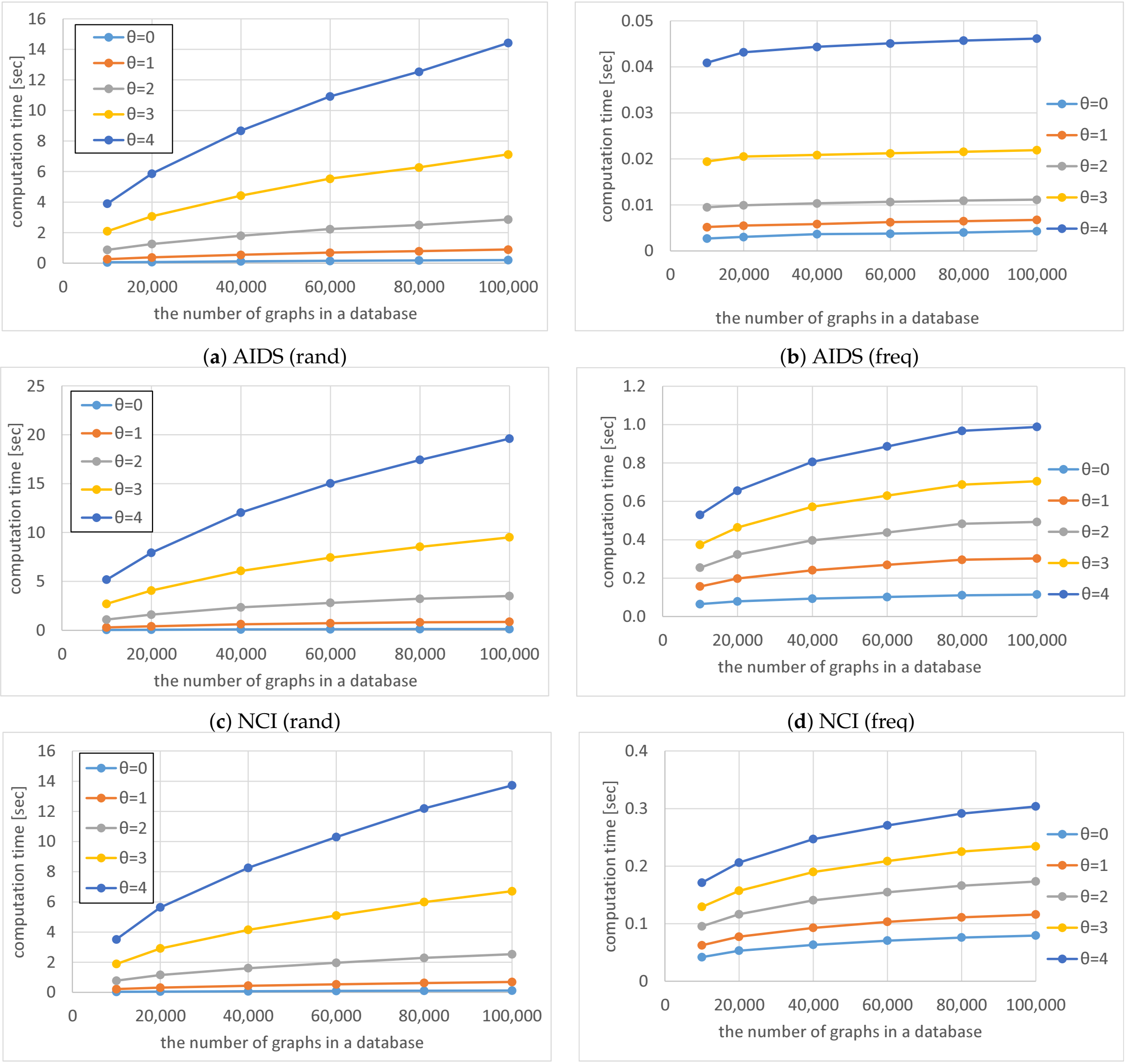
Table 2. Summary of datasets used in our experiments.

	AIDS		NCI		PubChem	
	Rand	**Freq**	**Rand**	**Freq**	**Rand**	**Freq**
# of vertex labels	31	10	42	15	16	9
# of edge labels	3	3	3	3	3	2
graphs in databases						
# of graphs	100,000	100,000	100,000	100,000	100,000	100,000
avg. # of vertices	29.0	21.4	28.2	16.9	27.6	24.1
max. # of vertices	100	26	79	27	84	32
min. # of vertices	1	10	1	5	1	11
avg. # of edges	30.0	20.4	29.2	16.1	28.2	23.1
query graphs						
# of graphs	100		100		100	
avg. # of vertices	71.0		63.9		64.9	
max. # of vertices	222		132		175	
min. # of vertices	42		40		34	
avg. # of edges	74.9		68.9		67.8	

Since the problem of similar supergraph search is a novel problem proposed in this paper, there are no methods for solving this problem other than our proposed method to the best of our knowledge. Although we cannot compare our proposed method with other methods, our method is based on CodeTree (We use "CodeTree" to name the method proposed in [17] and "code tree" to represent the index used in CodeTree). This is proposed in [17] and our method with $\theta = 0$ is equivalent to CodeTree. Our proposed method was implemented in Java and the experiments reported in this section were conducted on a workstation with an AMD Ryzen Threadripper 3970X 3.7 GHz CPU and 64 GB main memory.

First, we conducted experiments for Algorithm 4 with Equation (17). Figures 7–9 show the average computation time (query processing time) t, the average number of nodes n in the code tree that our method traversed, and the average number of solutions $|S|$, respectively, for various numbers of graphs $|G|$ in a database and various thresholds θ. The computation time increased as the number of graphs in a database increased, as shown in Figure 7. Although the time complexity of Algorithms 1 and 2 is linearly related to the number of graphs in the database, the computation time of our method was sublinear for the number of graphs in the database because Algorithm 4 maintains the correspondence between a vertex of a query graph and vertices of multiple graphs in the database simultaneously. In contrast, the computation time increased exponentially as θ was increased, as shown in Figure 7. This is because the number of graphs reachable by editing a graph $q' \subseteq q$ θ times increases exponentially. Although the computation time increased as the database size increased for *freq* databases, the rate of increase was less than for *rand* databases. This is because the numbers of nodes in code trees for *freq* databases are smaller than those for *rand* databases and the former numbers are relatively unchanged by the increase in database size, as shown in Figure 10. The reason why the numbers of nodes in code trees are largely unaffected by database size is that frequent subgraphs enumerated from the datasets are similar to each other and the AcGM codes of similar frequent subgraphs have common prefixes. This is because a subgraph of a frequent subgraph is also a frequent subgraph, which is a consequence of the anti-monotonic property of the support value [24].

As shown in Figure 8, the number of nodes in the code tree that Algorithm 4 traversed also increased exponentially as θ increased. The shapes of the curves plotted in Figure 8 are similar to those in Figure 7 and the trend for the numbers $|G|$ of graphs in databases and thresholds θ in Figure 8 are also similar to those in Figure 7. Figure 11 shows t/n for various numbers of graphs $|G|$ in a database and various thresholds θ. The figure shows that t/n is almost constant and unaffected by the number of graphs in a database, which indicates that the computation time of our method is proportional to the number of nodes in the code tree that Algorithm 4 traverses.



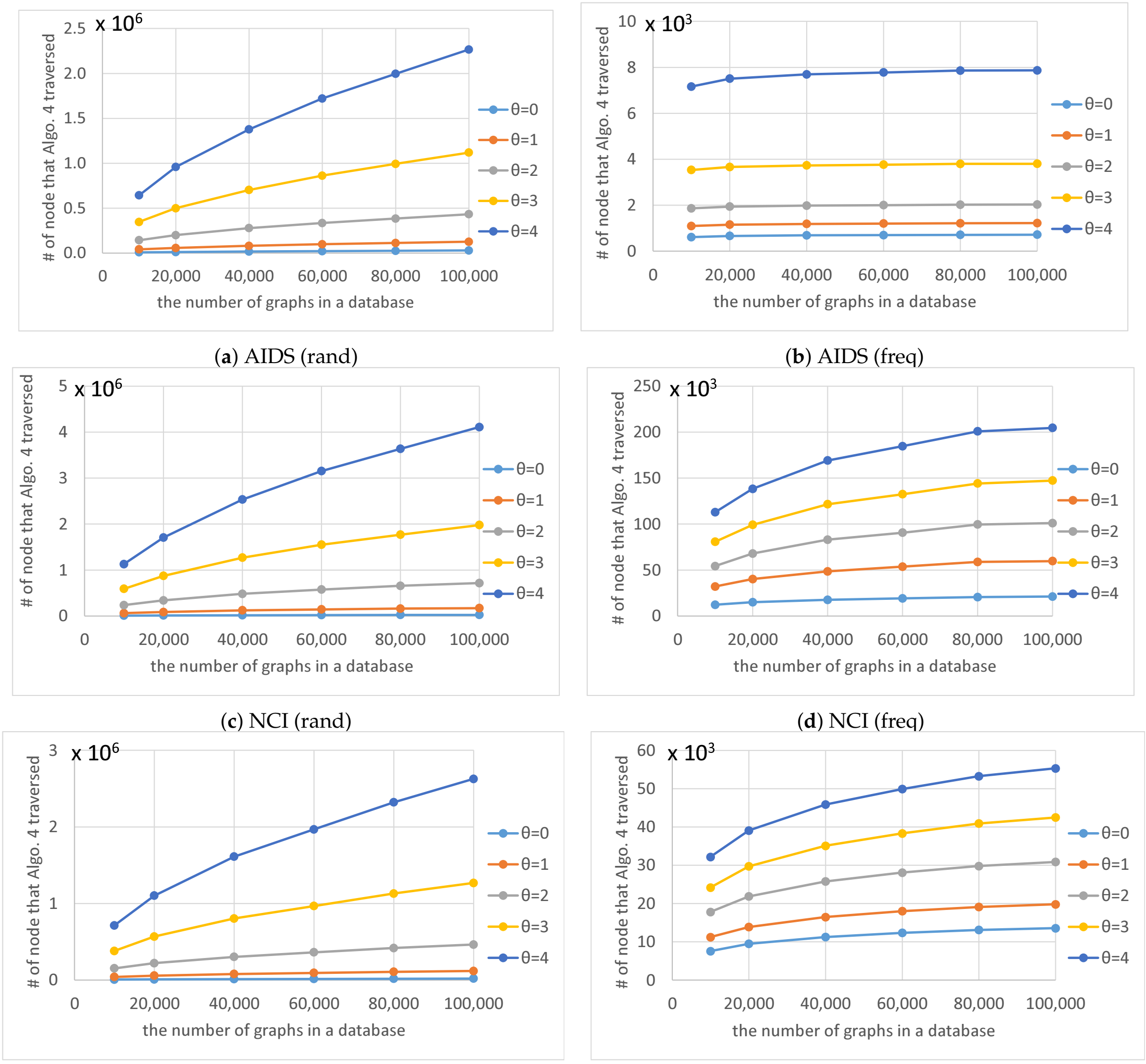
(**a**) AIDS (rand) (**b**) AIDS (freq)

(**c**) NCI (rand) (**d**) NCI (freq)

(**e**) PubChem (rand) (**f**) PubChem (freq)

Figure 7. Average computation times t for various numbers of graphs $|G|$ in a database and various thresholds θ.

As shown in Figure 9, the number of solutions found by Algorithm 4 was proportional to the number of graphs in the databases. Because there is a limit to the number of graphs in the database, the number of solutions did not increase exponentially as $|G|$ increased. The rectangle in Figure 12 shows the space in which graphs in a database are distributed and each point in the rectangle shows a graph in the database. Points inside the circle are solutions for a given query q and threshold θ. If the number of graphs in database G is doubled to G', such that $2|G| = |G'|$, the number of points surrounded by the circle doubles, if the distribution is not changed such that $\mathcal{P}(G) = \mathcal{P}(G')$. Thus, the number of solutions found by Algorithm 4 is proportional to the numbers of graphs in the databases.



(**a**) AIDS (rand)

(**b**) AIDS (freq)

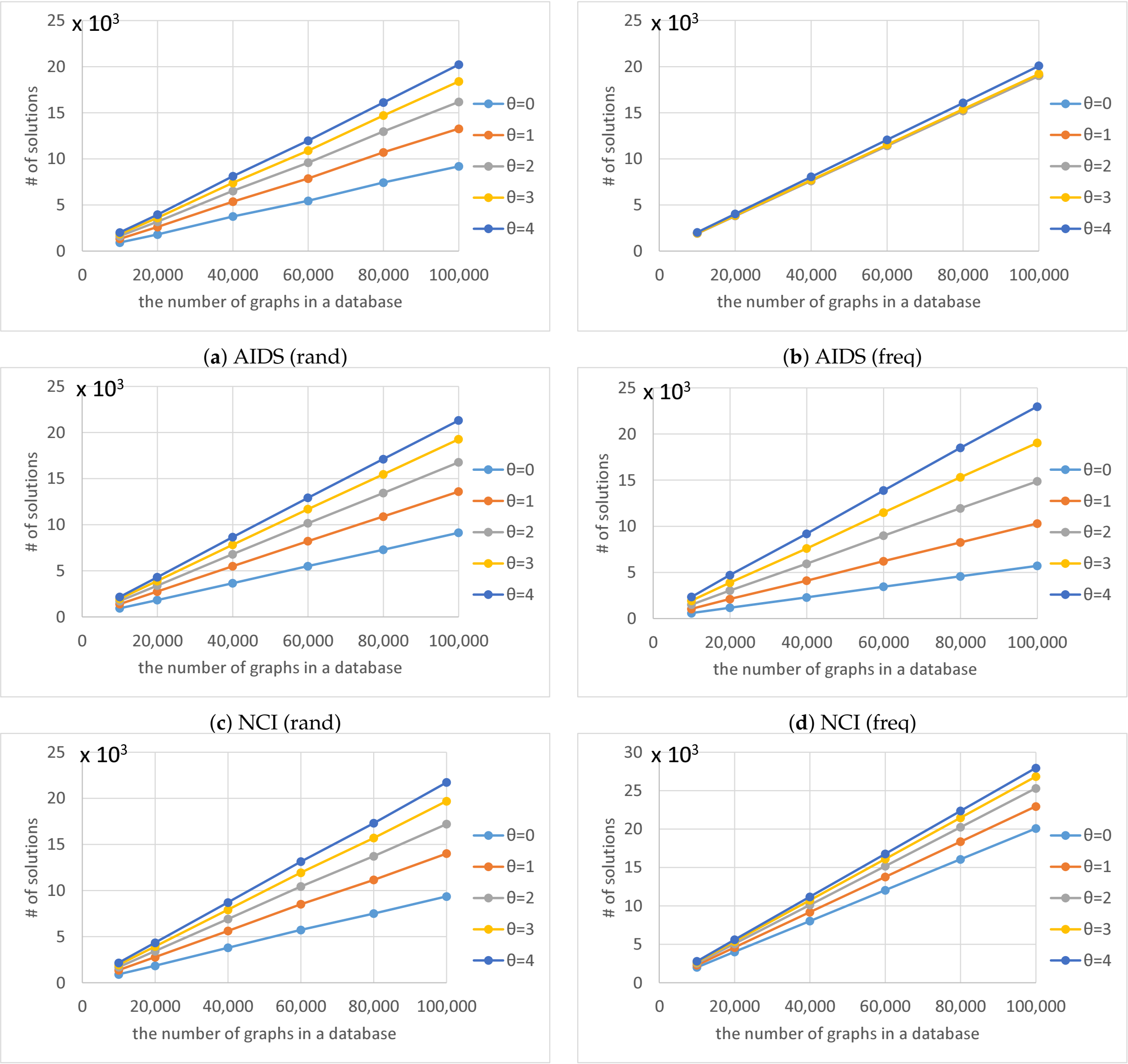
(**c**) NCI (rand)

(**d**) NCI (freq)

(**e**) PubChem (rand)

(**f**) PubChem (freq)

Figure 8. Average numbers of nodes n in the code tree that our method traversed for various numbers of graphs $|G|$ in a database and various thresholds θ.

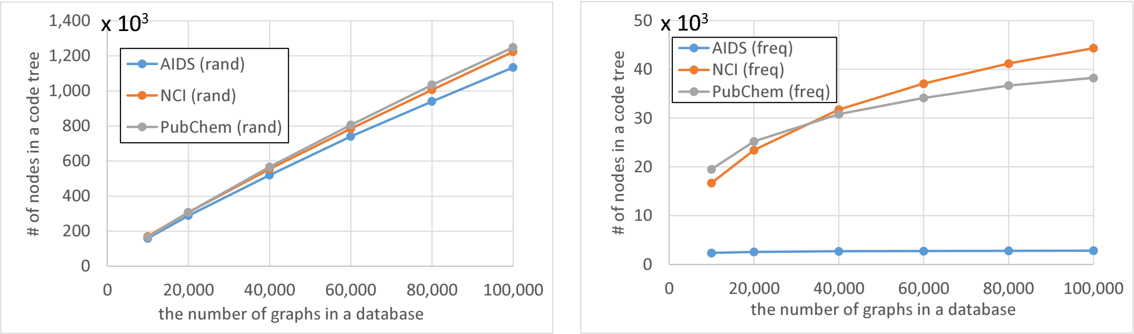


(**a**) AIDS (rand) (**b**) AIDS (freq)

(**c**) NCI (rand) (**d**) NCI (freq)

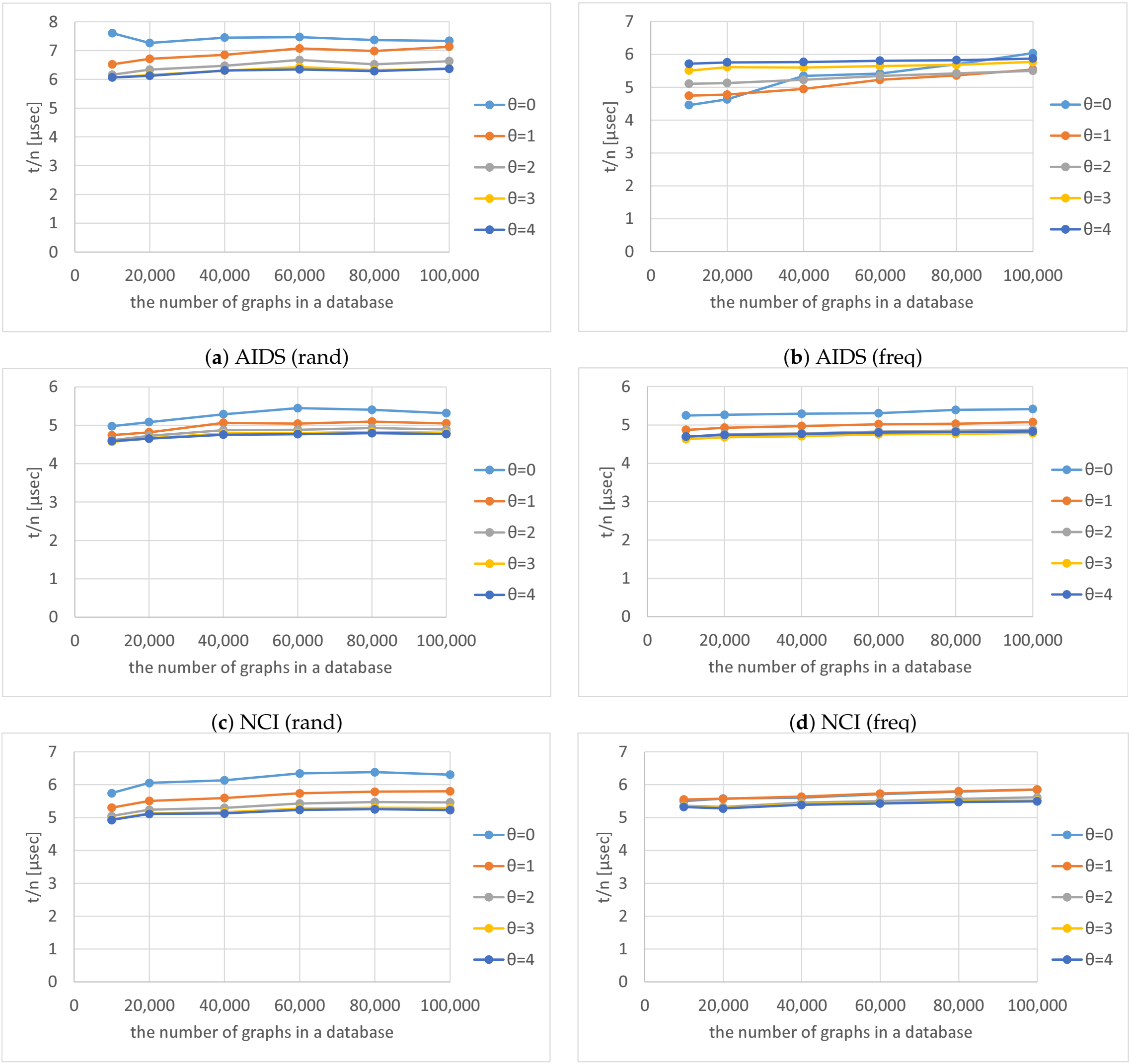
(**e**) PubChem (rand) (**f**) PubChem (freq)

Figure 9. Number of solutions $|S|$ for various numbers of graphs $|G|$ in a database and various thresholds θ.



(**a**) AIDS (rand) (**b**) AIDS (freq)

Figure 10. Number of nodes in code trees.



(**a**) AIDS (rand) (**b**) AIDS (freq)

(**c**) NCI (rand) (**d**) NCI (freq)

(**e**) PubChem (rand) (**f**) PubChem (freq)

Figure 11. Computation time per node t/n for various numbers of graphs $|G|$ in a database and various thresholds θ.

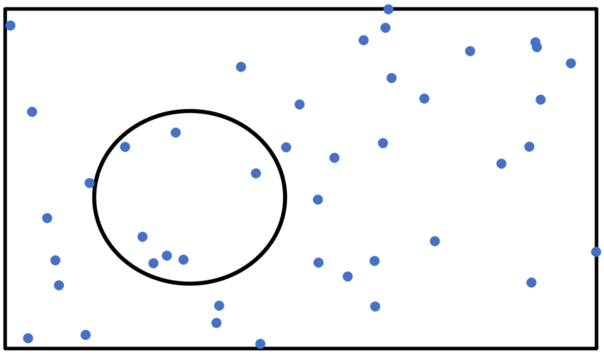
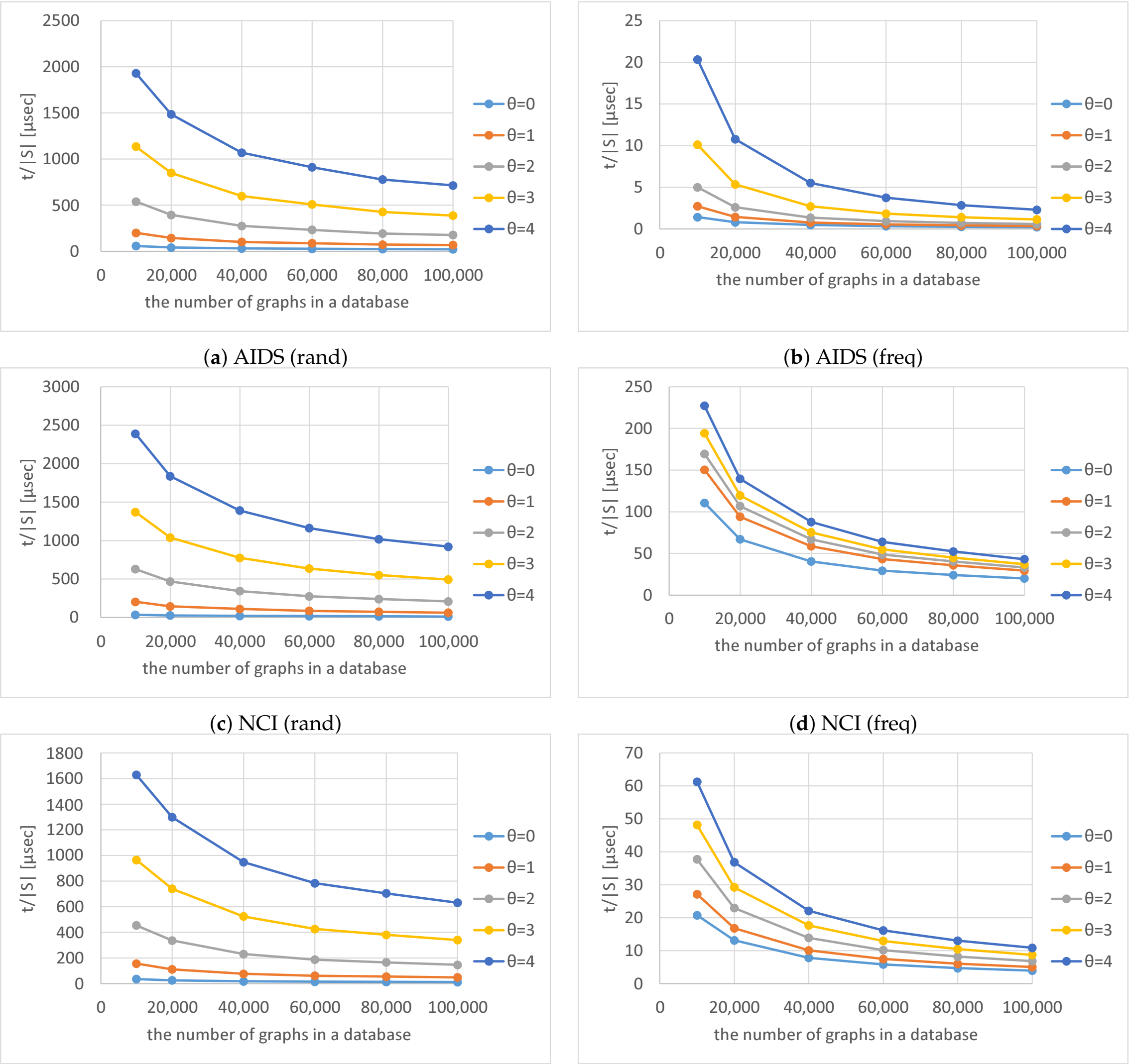


Figure 12. Distribution of graphs in a database.

Figure 13 shows $t/|S|$ for various numbers of graphs $|G|$ in a database and various thresholds θ; the figure indicates the average computation time to output each solution. As the numbers of graphs in the databases increased, $t/|S|$ decreased exponentially. This is because $|S|$ is proportional to the number of graphs in a database, whereas the overall computation time to search for solutions is sublinear.



(**a**) AIDS (rand)

(**b**) AIDS (freq)

(**c**) NCI (rand)

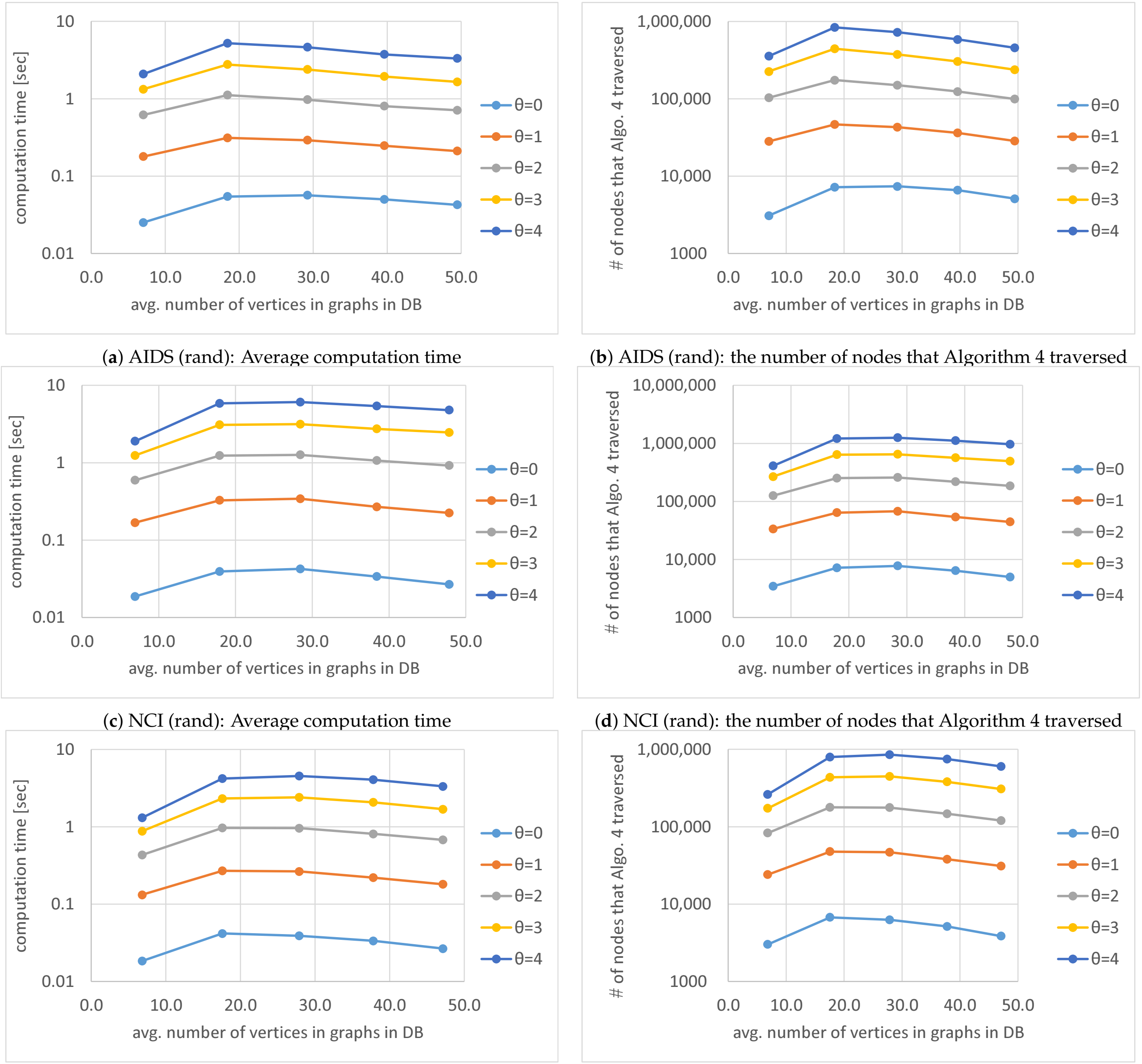
(**d**) NCI (freq)

(**e**) PubChem (rand)

(**f**) PubChem (freq)

Figure 13. Computation time per solution $t/|S|$ for various numbers of graphs $|G|$ in a database and various thresholds θ.

Figure 14 shows that average computation time and the number of nodes that Algorithm 4 traversed for various average numbers of vertices in graphs in a database and various thresholds θ. When the average number of vertices in graphs in a database is increased, the average computation time of Algorithm 4 is largely unaltered because the computation time of Algorithm 4 is proportional to the number of nodes that Algorithm 4 traverses, as mentioned after Algorithm 3. The number is also largely unaltered with respect to the change of the average number of the vertices. The reason why the number of nodes that Algorithm 4 traversed is largely unaltered is because of the use of the function $ed(s, s')$ shown in Equation (17). By using Equation (17), the relabeling of vertices or edges is admissible, whereas insertions and deletions of the vertices and edges are not admissible. Therefore, nodes that Algorithm 4 traverses are restricted. As mentioned in Section 7, one of the advantages of our proposed method is the customizability for searching for the types of graphs desired by users. By customizing the search for the types of graphs, the users can reduce not only the number of solutions to be outputted but also computation time of our proposed method.



(**a**) AIDS (rand): Average computation time

(**b**) AIDS (rand): the number of nodes that Algorithm 4 traversed

(**c**) NCI (rand): Average computation time

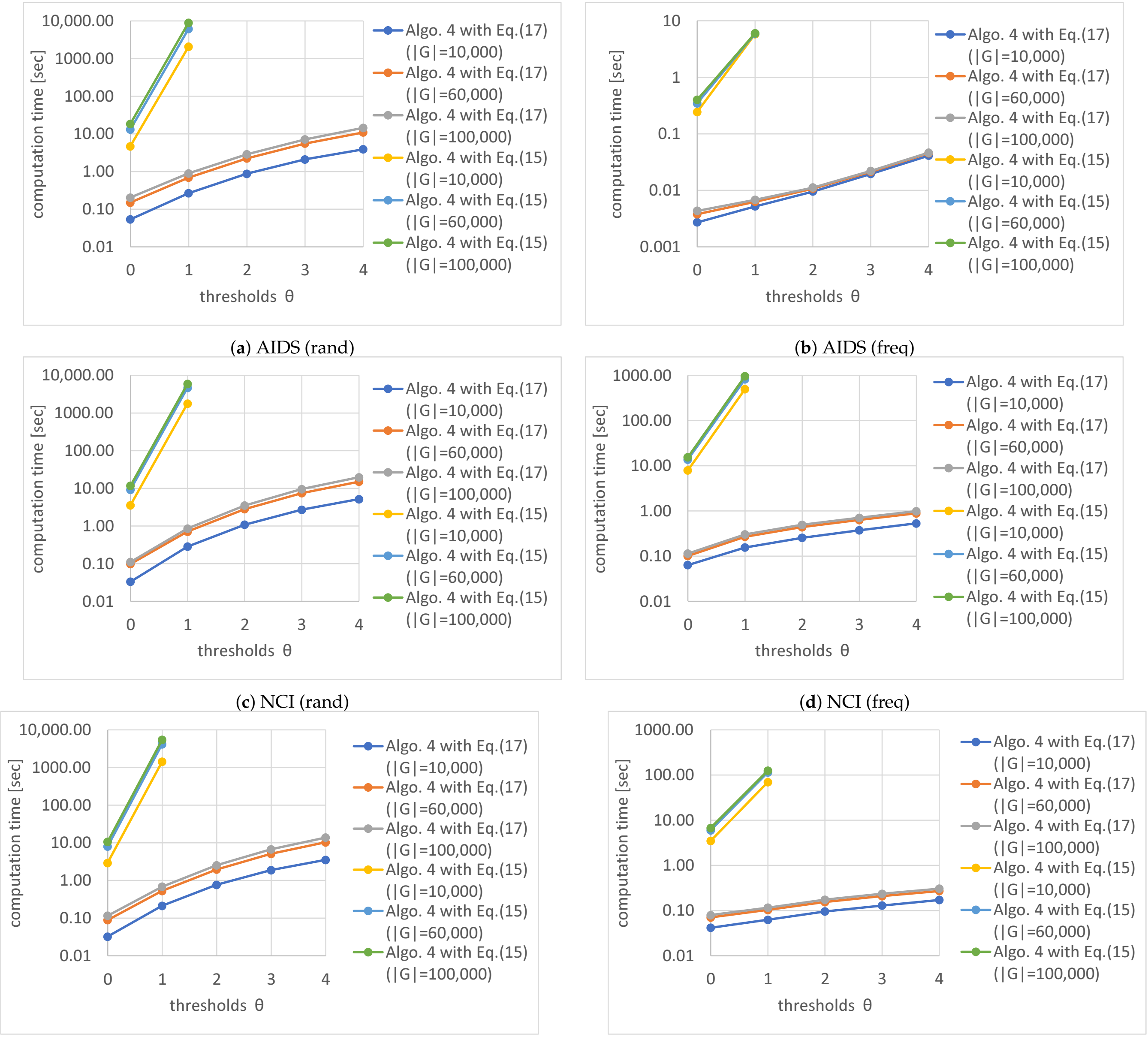
(**d**) NCI (rand): the number of nodes that Algorithm 4 traversed

(**e**) PubChem (rand): Average computation time

(**f**) PubChem (rand): the number of nodes that Algorithm 4 traversed

Figure 14. Average computation time and the average number of nodes that Algorithm 4 traversed for various numbers of vertices in graphs in a database and various thresholds θ.

Next, we conducted experiments for Algorithm 4 with Equations (15) and (17). Figure 15 shows the computation times for various numbers of graphs in the databases and various thresholds. The computation times with Equation (17) were much shorter than those with Equation (15). This is because the number of graphs reachable by editing a graph $q' \subseteq q$ θ times increases exponentially and editing graphs with Equation (17) is limited to the relabeling of vertices and edges, which is faster than the insertion and deletion of vertices and edges that are required with Equation (15).



(**a**) AIDS (rand)

(**b**) AIDS (freq)

(**c**) NCI (rand)

(**d**) NCI (freq)

(**e**) PubChem (rand)

(**f**) PubChem (freq)

Figure 15. Computation times of Algorithm 4 with Equation (15) and Equation (17) for various numbers of graphs $|G|$ in a database and various thresholds θ.

9. Conclusions

In this paper, we proposed a novel problem, which is called similar supergraph search, and designed an efficient algorithm to solve the problem. The problem is to identify all graphs in a database that are similar to any subgraph of a query graph, where similarity is defined as the edit distance. The proposed algorithm has three advantages for searching a database consisting of graphs for similar graphs. The first advantage is that the algorithm checks whether multiple graphs in the database are simultaneously solutions, because each node in the code tree is associated with the common prefix of the AcGM codes of multiple graphs in the database. The second advantage is that the codes produced by our proposed algorithm are limited to prefixes of the AGM codes of q and we do not need to solve the combination problem among the edges of q. The third advantage is the customization that enables users to search for their desired type of similar graphs in a database containing graphs. Therefore, the algorithm is a general and flexible framework for searching for similar subgraphs of query graphs and it is easily customizable for searching for the types of graphs desired by the user. Our experiments showed that the computation time increased exponentially as the distance threshold increased, but increased sublinearly with the number of graphs in the database.

Author Contributions: Conceptualization, A.I.; methodology, A.I.; software, M.Y. and A.I.; validation, M.Y. and A.I.; writing—original draft, A.I.; writing—review and editing, M.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by JSPS KAKENHI Grant Number JP20K11835.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not available.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Shiraki, K. Characteristics of a candidate of an antiviral medication against COVID-19. *Jpn. Med J.* **2020**, *5005*, 25–31. (In Japanese)
2. Bonnici, V.; Ferro, A.; Giugno, R.; Pulvirenti, A.; Shasha, D.E. Enhancing Graph Database Indexing by Suffix Tree Structure. In Proceedings of the IAPR International Conference on Pattern Recognition in Bioinformatics, Nijmegen, The Netherlands, 22–24 September 2010; pp. 195–203.
3. Cheng, J.; Ke, Y.; Ng, W.; Lu, A. FG-Index: Towards Verification-Free Query Processing on Graph Databases. In Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, 11–14 June 2007; pp. 857–872.
4. Cheng, J.; Ke, Y.; Ng, W. Efficient Query Processing on Graph Databases. *ACM Trans. Database Syst.* **2009**, *2*, 48.
5. Klein, K.; Kriege, N.M.; Mutzel, P. CT-Index: Fingerprint-based Graph Indexing Combining Cycles and Trees. In Proceedings of the IEEE International Conference on Data Engineering, Hannover, Germany, 11–16 April 2011; pp. 1115–1126.
6. Shang, H.; Zhang, Y.; Lin, X.; Yu, J.X. Taming Verification Hardness: an Efficient Algorithm for Testing Subgraph Isomorphism. *Proc. Vldb Endow.* **2008**, *1*, 364–375.
7. Sun, S.; Luo, Q. Scaling Up Subgraph Query Processing with Efficient Subgraph Matching. In Proceedings of the IEEE International Conference on Data Engineering, Paris, France, 16–19 April 2019; pp. 220–231.
8. Williams, D.W.; Huan, J.; Wang, W. Graph Database Indexing Using Structured Graph Decomposition. In Proceedings of the IEEE International Conference on Data Engineering, Istanbul, Turkey, 17–20 April 2007; pp. 976–985.
9. Xie, Y.; Yu, P.S. CP-Index: on the Efficient Indexing of Large Graphs. In Proceedings of the ACM Conference on Information and Knowledge Management, Glasgow, UK, 24–28 October 2011; pp. 1795–1804.
10. Yan, X.; Yu, P.S.; Han, J. Graph Indexing: A Frequent Structure-based Approach. In Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, 13–18 June 2004; pp. 335–346.
11. Yuan, D.; Mitra, P. Lindex: A Lattice-based Index for Graph Databases. *VLDB J.* **2013**, *22*, 229–252.
12. Zhang, S.; Hu, M.; Yang, J. TreePi: A Novel Graph Indexing Method. In Proceedings of the IEEE International Conference on Data Engineering, Istanbul, Turkey, 17–20 April 2007; pp. 966–975.
13. Zhao, P.; Yu, J.X.; Yu, P.S. Graph Indexing: Tree + Delta >= Graph. In Proceedings of the International Conference on Very Large Data Bases, Vienna, Austria, 23–27 September 2007; pp. 938–949.
14. Zou, L.; Chen, L.; Yu, J.X.; Lu, Y. A Novel Spectral Coding in a Large Graph Database. In Proceedings of the International Conference on Extending Database Technology, Nantes, France, 25–29 March 2008; pp. 181–192.
15. Chen, C.; Yan, X.; Yu, P.S.; Han, J.; Zhang, D.; Gu, X. Towards Graph Containment Search and Indexing. In Proceedings of the International Conference on Very Large Data Bases, Vienna, Austria, 23–27 September 2007; pp. 926–937.
16. Cheng, J.; Ke, Y.; Fu, A.W.; Yu, J.X.: Fast Graph Query Processing with a Low-Cost Index. *VLDB J.* **2011**, *20*, 521–539.
17. Imai, S.; Inokuchi, A. Efficient Supergraph Search Using Graph Coding. *IEICE Trans. Inf. Syst.* **2020**, *103-D(1)*, 130–141.
18. Kim, H.; Min, S.; Park, K.; Lin, X.; Hong, S.; Han, W. IDAR: Fast Supergraph Search Using DAG Integration. *Proc. Vldb Endow.* **2020**, *13*, 1456–1468.
19. Lyu, B.; Qin, L.; Lin, X.; Chang, L.; Yu, J.X. Scalable Supergraph Search in Large Graph Databases. In Proceedings of the IEEE International Conference on Data Engineering, Helsinki, Finland, 16–20 May 2016; pp. 157–168.
20. Yuan, D.; Mitra, P.; Giles, C.L. Mining and Indexing Graphs for Supergraph Search. *Proc. Vldb Endow.* **2013**, *6*, 829–840.
21. Zhang, S.; Li, J.; Gao, H.; Zou, Z. A Novel Approach for Efficient Supergraph Query Processing on Graph Databases. In Proceedings of the International Conference on Extending Database Technology, Saint-Petersburg, Russia, 24–26 March 2009; pp. 204–215.
22. Zhu, G.; Lin, X.; Zhang, W.; Wang, W.; Shang, H. PrefIndex: An Efficient Supergraph Containment Search Technique. In Proceedings of the International Conference on Scientific and Statistical Database Management, Heidelberg, Germany, 30 June–2 July 2010; pp. 360–378.

23. Riesen, K. Structural Pattern Recognition with Graph Edit Distance—Approximation Algorithms and Applications. In *Advances in Computer Vision and Pattern Recognition*; Springer: Berlin, Germany, 2015.
24. Inokuchi, A.; Washio, T.; Motoda, H. An Apriori-based Algorithm for Mining Frequent Substructures from Graph Data. In Proceedings of the European Conference on Principles of Data Mining and Knowledge Discovery, Lyon, France, 13–16 September 2000; pp. 13–23.
25. Chang, L.; Feng, X.; Lin, X.; Qin, L.; Zhang, W.; Ouyang, D. Speeding Up GED Verification for Graph Similarity Search. In Proceedings of the IEEE International Conference on Data Engineering, Dallas, TX, USA, 20–24 April 2020; pp. 793–804.
26. Gouda, K.; Hassaan, M. CS_GED: An Efficient Approach for Graph Edit Similarity Computation. In Proceedings of the IEEE International Conference on Data Engineering, Helsinki, Finland, 16–20 May 2016; pp. 265–276.
27. Kim, J.; Choi, D.; Li, C. Inves: Incremental Partitioning-based Verification for Graph Similarity Search. In Proceedings of the International Conference on Extending Database Technology, Lisbon, Portugal, 26–29 March 2019; pp. 229–240.
28. Liang, Y.; Zhao, P. Similarity Search in Graph Databases: A Multi-Layered Indexing Approach. In Proceedings of the IEEE International Conference on Data Engineering, San Diego, CA, USA, 19–22 April 2017; pp. 783–794.
29. Wang, X.; Ding, X.; Tung, A.K.H.; Ying, S.; Jin, H. An Efficient Graph Indexing Method. In Proceedings of the IEEE International Conference on Data Engineering, Arlington, VA, USA, 1–5 April 2012; pp. 210–22.
30. Zhao, X.; Xiao, C.; Lin, X.; Wang, W.; Ishikawa, Y. Efficient Processing of Graph Similarity Queries with Edit Distance Constraints. *VLDB J.* **2013**, *22*, 727–752.
31. Zhao, X.; Xiao, C.; Lin, X.; Zhang, W.; Wang, Y. Efficient Structure Similarity Searches: A Partition-based Approach. *VLDB J.* **2018**, *27*, 53–78.
32. Zheng, W.; Zou, L.; Lian, X.; Wang, D.; Zhao, D. Efficient Graph Similarity Search Over Large Graph Databases. *IEEE Trans. Knowl. Data Eng.* **2015**, *27*, 964–978.
33. Inokuchi, A.; Washio, T.; Nishimura, Y.; Motoda, H. *A Fast Algorithm for Mining Frequent Connected Subgraphs*; IBM Research: Yorktown Heights, NY, USA, 2002.
34. Yan, X.; Han, J. gSpan: Graph-Based Substructure Pattern Mining. In Proceedings of the IEEE International Conference on Data Mining, Maebashi City, Japan, 9–12 December 2002; pp. 721–724.
35. Bi, F.; Chang, L.; Lin, X.; Qin, L.; Zhang, W. Efficient Subgraph Matching by Postponing Cartesian Products. In Proceedings of the International Conference on Management of Data, San Francisco, CA, USA, 26 June–1 July 2016; pp. 1199–1214.
36. Sun, Z.; Wang, H.; Wang, H.; Shao, B.; Li, J. Efficient Subgraph Matching on Billion Node Graphs. *Proc. Vldb Endow.* **2012**, *5*, 788–799.
37. Zhang, S.; Li, S.; Yang, J. GADDI: Distance Index based Subgraph Matching in Biological Networks. In Proceedings of the International Conference on Extending Database Technology, Saint Petersburg, Russia, 24–26 March 2009; pp. 192–203.
38. Khan, A.; Li, N.; Yan, X.; Guan, Z.; Chakraborty, S.; Tao, S. Neighborhood based Fast Graph Search in Large Networks. In Proceedings of the ACM SIGMOD International Conference on Management of Data, Athens, Greece, 12–16 June 2011; pp. 901–912.
39. Khan, A.; Wu, Y.; Aggarwal, C.C.; Yan, X. NeMa: Fast Graph Search with Label Similarity. *Proc. Vldb Endow.* **2013**, pp. 181–192.
40. Tian, Y.; McEachin, R.C.; Santos, C.; States, D.J.; Patel, J.M. SAGA: A Subgraph Matching Tool for Biological Graphs. *Bioinformatics* **2007**, *23*, 232–239.
41. Zhang, S.; Yang, J.; Jin, W. SAPPER: Subgraph Indexing and Approximate Matching in Large Graphs. *Proc. Vldb Endow.* **2010**, *3*, 1185–1194.
42. Borgwardt, K.M.; Ghisu, M.E.; Llinares-López, F.; O'Bray, L.; Rieck, B. Graph Kernels: State-of-the-Art and Future Challenges. *Found. Trends Mach. Learn.* **2002**, *13*, 531–712.
43. Wang, X.; Smalter, A.M.; Huan, J.; Lushington, G.H. G-Hash: Towards Fast Kernel-based Similarity Search in Large Graph Databases. In Proceedings of the International Conference on Extending Database Technology, Nantes, France, 25–29 March 2008; pp. 472–480.
44. Raymond, J.W.; Willett, P. Maximum Common Subgraph Isomorphism Algorithms for the Matching of Chemical Structures. *J. Comput. Aided Mol. Des.* **2002**, *16*, 521–533.
45. Bahiense, L.; Manic, G.; Piva, B.; de Souza, C.C. The Maximum Common Edge Subgraph Problem: A Polyhedral Investigation. *Discret. Appl. Math.* **2012**, *160*, 2523–2541.
46. Kashima, H.; Tsuda, K.; Inokuchi, A. Marginalized Kernels Between Labeled Graphs. In Proceedings of the International Conference on Machine Learning, Washington, DC, USA, 21–24 August 2003; pp. 321–328.
47. Shervashidze, N.; Schweitzer, P.; van Leeuwen, E.J.; Mehlhorn, K.; Borgwardt, K.M. Weisfeiler-Lehman Graph Kernels. *J. Mach. Learn. Res.* **2011**, *12*, 2539–2561.