Inves: Incremental Partitioning-Based Verification for Graph Similarity Search

Jongik Kim
Chonbuk National University
Jeonju, Republic of Korea
jongik@jbnu.ac.kr

Dong-Hoon Choi
KISTI
Daejeon, Republic of Korea
choid@kisti.re.kr

Chen Li
University of California
Irvine, CA
chenli@ics.uci.edu

ABSTRACT

We study the problem of graph similarity search with a graph edit distance (GED) constraint. Existing solutions adopt a filtering-and-verification framework, with a focus on the filtering phase where a feature-based index is used to reduce the number of candidate graphs to be verified. These solutions suffer from a computationally expensive verification phase. In this paper, we develop a novel technique called Inves that can significantly reduce the time of verifying a candidate graph. Its main idea is to judiciously and incrementally partition a candidate graph based on the query graph, and use the results to compute a lower bound of their distance. If a full GED computation is needed, Inves utilizes the collected information, and uses novel methods and an A* algorithm to search in the space of possible vertex mappings between the graphs to compute their GED efficiently. A main advantage of Inves is that it can be adopted by a plethora of graph similarity search algorithms. Our extensive experiments on both real and synthetic datasets show that Inves can significantly improve the performance of existing techniques by an order of magnitude.

1 INTRODUCTION

Graph data models are widely used in representing complex objects, such as chemical compounds, social networks, and biological structures. Graph search, which finds all occurrences of a query graph in a database of graphs, is a fundamental operation needed in many applications. To tolerate data inconsistency, natural noises, and different data representations in graph search, very often these applications require finding graphs similar to a given query graph. Various similarity measures have been proposed, such as maximum common subgraphs [2, 15], missing edges and features [23, 28], and graph alignment [17]. Among them, one of the commonly used metric is graph edit distance (GED) [5, 6, 22], which can capture the structural difference between graphs, and can be applied to many types of graphs [22, 24]. The GED between two graphs is the minimum number of graph edit operations to transform one to the other, where a graph edit operation is insertion, deletion, or substitution of a single vertex or edge.

The problem of graph similarity search is to find graphs in a database whose GED to a query graph is within a given threshold. This problem is challenging because GED computation between two graphs is NP-hard [22]. Generally, a scan-based approach that directly computes the GED between each data graph and the query graph is computationally prohibitive. Many existing solutions adopt a filtering-and-verification framework. An index structure is typically used to generate candidate graphs in the filtering phase, and each candidate is compared with the query graph to find if it is a true match in the verification phase. Existing studies mainly focus on developing a feature-based index to generate candidates in the filtering phase. For example, c-star [22] and k-AT [19] extract tree-structured features from data graphs and build an inverted index on the extracted features. GSimSearch [25, 26] builds an inverted index on path-based features of graphs. Pars [24] and MLIndex [12] utilize partitions of graphs as features to be indexed.

The performance of existing solutions can suffer from too many candidates and an expensive verification phase. Table 1 shows the performance of 100 queries using one of the index-based search algorithms, Pars [24], on an AIDS dataset containing 42,687 graphs (see Section 5 for details). In the table, we use the number of data graphs that have passed a primitive filter named the global label filter (refer to [25] and Section 3.5 for details of the filter). The number of candidates denotes those candidates that require full GED computations. For example, when the threshold $\tau = 5$, only 15.5% of data graphs are filtered from the index-based filtering phase. Experiments on other solutions show similar behaviors.

Table 1: Performance of Pars

GED threshold τ	1	2	3	4	5
# of data graphs	574	3,335	12,669	34,774	74,937
# of candidates	142	591	4,931	22,846	63,301
# of answers	105	135	161	221	278
Filtering ratio	75.3%	82.3%	61.1%	34.3%	15.5%

To solve this problem, in this paper we develop a novel verification technique, called Inves[1]. Given a set of candidate graphs generated from a filtering phase, the proposed technique can effectively reduce the time for verifying if the GED between each candidate graph and the query graph is within a given threshold. Its main idea is to judiciously and incrementally partition the candidate graph based on the query graph, and use the results to try to prune this pair. If a full GED computation is needed, Inves utilizes the collected information, and uses novel methods and an A* algorithm to search in the space of possible vertex mappings between the graphs to compute their GED efficiently. A main advantage of Inves is that it can be adopted by a plethora of graph similarity search algorithms.

The following are our contributions:

- We propose Inves as a novel incremental partitioning-based verification technique. Given a candidate graph and a query graph with a GED threshold, Inves incrementally isolates subgraphs of the candidate graph that cause mismatches with the query graph. If the number of isolated subgraphs is greater than

[1] It stands for Incremental partitioning-based verification technique for graph edit similarity search.

the threshold, Inves filters out this pair since their GED cannot be within the threshold. In Section 3, we present the details of the incremental partitioning-based verification framework.

- If the pair of graphs cannot be pruned using the generated subgraphs, Inves employs efficient methods based on a well-known A* algorithm for GED computation [14]. It also takes advantage of the partitioning results by first considering those vertices that cause edit errors. In this way, it can significantly reduce the search space of the A* algorithm, thus improve the performance of GED computation (Section 4).
- We conduct extensive experiments to evaluate Inves on both real and synthetic data sets (Section 5). The results show the benefits of the various optimization methods in the technique. In addition, by adopting Inves in existing index-based algorithms, we can significantly reduce the total running time by an order of magnitude.

The rest of the paper is organized as follows: Section 2 provides preliminaries and reviews related work. Section 3 presents the proposed verification framework, and Section 4 provides our GED computation methods. Section 5 presents experimental results, and Section 6 concludes the paper.

2 PRELIMINARIES AND RELATED WORK

2.1 Graph Similarity Search Problem

We focus on undirected labeled simple graphs defined as follows. An undirected labeled simple graph g is a triple (V_g, E_g, L_g), where V_g is a set of vertices, $E_g \subseteq \{(u, v) \mid u \in V_g \wedge v \in V_g \wedge u \neq v\}$ is a set of edges, and L_g is a labeling function that maps vertices and edges to labels. $L_g(v)$ and $L_g(u, v)$ respectively denote the label of a vertex v and the label of an edge (u, v). If there is no edge between u and v, $L_g(u, v)$ returns a unique value λ distinguished from all other edge labels. There are no self-loops nor more than one edge between two vertices. For simplicity, in the rest of the paper, we use graph to denote undirected labeled simple graph.

The graph edit distance (or GED for short) between two graphs x and y, denoted by $ged(x, y)$, is the minimum number of graph edit operations that transform x to y. A graph edit operation is one of the following: (1) insertion of an isolated labeled vertex; (2) deletion of an isolated labeled vertex; (3) substitution of the label of a vertex; (4) insertion of a labeled edge; (5) deletion of a labeled edge; or (6) substitution of the label of an edge.

Example 1. *Figure 1 shows two graphs x and y, which include vertex labels representing atom symbols and edge labels (i.e., single and double lines) representing chemical bonds. Besides vertex labels, the graphs also include vertex identifiers. To transform x into y, we can do the following three graph edit operations on x: insertion of a single-bond edge between u_3 and u_5, and substitutions of labels of u_2 and u_8. Therefore, $ged(x, y)$ is 3.*

We formalize the problem of graph edit similarity search as follows.

Definition 1 (Graph Similarity Search Problem). *For a graph database $\mathcal{D} = \{x_1, \ldots, x_n\}$ and a query graph y with a GED threshold τ, the graph edit similarity search finds every data graph $x_i \in \mathcal{D}$ such that $ged(x_i, y) \leq \tau$.*

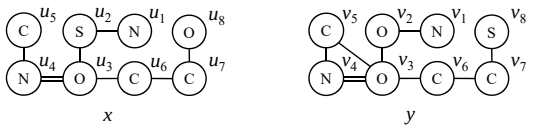


Figure 1: Two example graphs

2.2 A* algorithm for GED Computation

In this section, we review the most widely used algorithm for GED computation [14], which is based on A*. Given a pair of graphs x and y, the A* algorithm basically traverses all possible vertex mappings between x and y in a best-first fashion. It maintains a priority queue that contains states in its state-space tree, where each state in the tree represents a partial vertex mapping between the pair. The priority (or edit distance) of a state is determined by the sum of (1) the existing distance g: the edit operations detected from the initial state to the current state; and (2) an estimated distance h: a heuristic estimation of the edit operations from the current state to the goal. The A* algorithm guarantees that it finds an optimal mapping if h is not overestimated.

Algorithm 1: GED(x, y, τ)

input : x and y are graphs; τ is a GED threshold.
output: if $ged(x, y) \leq \tau$, $ged(x, y)$; otherwise, τ+1

```
1  O ← order the vertices in x;
2  Q ← ∅; Q.push(∅);
3  while Q ≠ ∅ do
4      M ← Q.pop();
5      if complete(M) then return existingDistance(M);
6      u ← next unmapped vertex in V_x ∪ {ε} as per O;
7      foreach v ∈ (V_y ∪ {ε}) s.t. v ∉ M do
8          g ← existingDistance(M ∪ {u → v});
9          h ← estimateDistance(M ∪ {u → v});
10         if g + h ≤ τ then Q.pushQueue(M ∪ {u → v});
11 return τ + 1;
```

The GED computation algorithm is outlined in Algorithm 1. It first determines the order of vertices in x, and pushes the initial state, i.e., an empty mapping, into the queue (Lines 1–2). In the main loop, it removes a mapping M from the queue that has a minimum edit distance (Line 4). If M contains all vertices of x and y, it returns the existing distance of M (Line 5). Otherwise, it expands its state-space tree by mapping the next unmapped vertex u in x (Line 6) to each unmapped vertex v in y (Line 7). It pushes each expanded state into the queue if the edit distance of the state is not greater than τ (Lines 8–10). In the algorithm, ε is used to denote an insertion or a deletion of a vertex. If it fails to find any mapping whose edit distance is not greater than τ, it returns $\tau + 1$ (Line 11).

2.3 Related Work

Previous work on the graph similarity search utilizes small overlapping substructures to establish a filtering condition between dissimilar graphs. Motivated by the gram idea used in string similarity searches, the k-AT algorithm [19] defines a q-gram as a tree rooted at a vertex v with all vertices reachable to v in q hops. A star structure, which is 1-gram defined by k-AT, has been proposed to set up a filtering condition through bipartite matching between star structures [22]. SEGOS [20] is a two-level index structure proposed to efficiently search star structures. The main focus of these approaches has been on the filtering phase to develop efficient index-based filtering methods using those substructures.

GSimSearch [25, 26] proposed a path-based q-gram and developed an index-based filtering technique based on the observation of the algorithm called ED-join [21] in string search. To further reduce the number of candidates, GSimSearch proposed local label filtering in its verification phase. However, this technique is based on small fixed-size substructures of graphs, thus edit errors are mainly captured from label differences, and structural differences are considered inside small substructures only.

There is recent work that makes use of large disjoint substructures of graphs to capture structural differences between graphs. Pars [24] partitions data graphs into disjoint subgraphs, and makes an index on the partitioned subgraphs. Using the index, it identifies data graphs having partitions contained in the query graph, and generates them as candidate graphs. It employs a random-graph-partitioning strategy and refines initial partitioning results based on a query workload. It also dynamically rearranges indexed partitions in a restricted way while searching its index structure. MLIndex [12] was proposed to reduce the number of candidates by indexing a few alternative partitioning results of data graphs. It defines a selectivity of a partition based on vertex and edge label frequencies, and divides a graph in a way to increase selectivities of partitions. Despite the efforts in the previous approaches, their filtering power of partitions is inherently limited because partitions of data graphs are determined offline, and one or a few rigid partitionings of a data graph cannot work well for all queries.

Other related work includes Mixed [27] and LBMatrix [3]. Mixed generates candidates by using small and large disjoint substructures of a query graph. LBMatrix has proposed a q-gram-based matrix index structure that can be stored in external memory to handle very large datasets.

3 INVES: VERIFICATION FRAMEWORK

In this section, we propose the Inves verification framework aiming to efficiently verify if the GED between a pair of graphs is within a given threshold. We first introduce the partition-based verification principle, then present the details of Inves.

3.1 Partition-based Verification Scheme

Due to the high cost of GED computation, it makes the graph similarity search impractical to directly compute the GED between a candidate and the query when there are many candidates generated from an index-based filtering phase. To efficiently verify a pair of graphs, in this paper we use a partition-based lower bound of the GED between the pair before computing the exact GED. We begin with the concept of an induced subgraph for defining graph partitions, then present the verification scheme.

Definition 2 (Induced Subgrpah Isomorphism). *A graph r is induced subgraph isomorphic to another graph s, denoted as $r \sqsubseteq s$, if there exists an injection $f: V_r \rightarrow V_s$ such that $\forall u \in V_r$, $f(u) \in V_s \wedge L_r(u) = L_s(f(u))$ and $\forall u \in V_r$, $\forall v \in V_r$, $L_r(u,v) = L_s(f(u), f(v))$. In this case, the graph r is called an induced subgraph of s.*

Recall that the edge labeling function $L_g(u,v)$ returns a unique value λ if there is no edge between u and v in a graph g. It enables us to check the inducedness of a subgraph in Definition 2.

Example 2. *Consider the graphs p_1, p_2, and y in Figure 2. $p_1 \sqsubseteq y$, but $p_2 \not\sqsubseteq y$ because $L_{p_2}(u_4, u_6) = \lambda \neq L_y(v_3, v_5) =$ single-bond.*

Given a graph g and a vertex set $V \subseteq V_g$, there is only one induced subgraph p of g such that $V_p = V$. That is, p is uniquely identified by V. Therefore, we use V interchangeably with the induced subgraph of g defined by V.

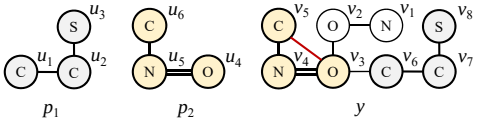


Figure 2: Induced subgraph isomorphism

Definition 3 (Graph Partitioning). *A partitioning of a graph g is $\mathcal{P}(g) = \{p_1, \ldots, p_k\}$ such that $\forall i\ p_i \sqsubseteq g$, $\forall i, j\ i \neq j \Rightarrow V_{p_i} \cap V_{p_j} = \emptyset$, and $V_g = \bigcup_{i=1}^{k} V_{p_i}$.*

Given a pair of graphs x and y, consider a partition $p \in \mathcal{P}(x)$. If $p \sqsubseteq y$, we say p is *matching* with y. Otherwise, we say p is *mismatching* with y. We also simply call p a matching (or mismatching) partition if y is clear from the context. An induced subgraph o of y such that $V_o \subseteq V_y$ is called an *occurrence* of p in y if and only if $p \sqsubseteq o$ and $o \sqsubseteq p$. In Figure 2, for example, $o = \{v_6, v_7, v_8\}$ is an occurrence of p_1 in y.

With the graph partitioning, a lower bound of the GED between a pair of graphs are calculated as follows.

Lemma 1. *Consider a pair of graphs x and y with a graph partitioning $\mathcal{P}(x)$. $lb(x,y) = |\{p \mid p \in \mathcal{P}(x) \wedge p \not\sqsubseteq y\}|$ is a lower bound of the GED between the pair.*

Proof. Since partitions of x share neither a vertex nor an edge, an edit operation on a partition does not affect to another partition. Therefore, each mismatching partition p requires at least one edit operation to transform x to y. □

The following corollary states the partition-based verification scheme based on the lower bound in Lemma 1.

Corollary 1. *Given a GED threshold τ, consider a pair of graphs x and y with a graph partitioning $\mathcal{P}(x)$. If $lb(x,y) > \tau$, the pair can be pruned without the GED computation.*

Partition-based lower bounds and their variants have been extensively studied and discussed in the literature of string similarity search (e.g. [8, 11]) and approximate subsequence mapping (e.g. [1, 9, 10]). The same principle is well adopted in recent work for graph similarity search [12, 24, 27]. Our lower bound in Lemma 1 is a simple extension of existing partition-based approaches. While the focus of existing work is on building a partition-based inverted index for the filtering phase, our focus in this paper is on the verification phase to efficiently verify a candidate graph using the partition-based lower bound.

To obtain the lower bound in Lemma 1, we need $|\mathcal{P}(x)|$ induced subgraph isomorphism tests, which are generally NP-hard. However, former studies have empirically showed that subgraph isomorphism test is on average three orders of magnitude faster than GED computation [12, 24], and thus it can be practically used in deriving a partition-based lower bound.

Example 3. *Consider a pair of graphs x and y shown in Figure 1 with a GED threshold $\tau = 1$. If we partition x into $\{p_1, p_2\}$ as depicted in Figure 3(a), $lb(x,y) = 2$ because $p_1 \not\sqsubseteq y$ and $p_2 \not\sqsubseteq y$. Therefore, we can safely prune the pair without GED computation according to Corollary 1. If we partition x into $\{p'_1, p'_2\}$ as illustrated in Figure 3(b), $lb(x,y) = 1$ since $p'_1 \not\sqsubseteq y$ but $p'_2 \sqsubseteq y$. Thus, we need a GED computation between the pair.*

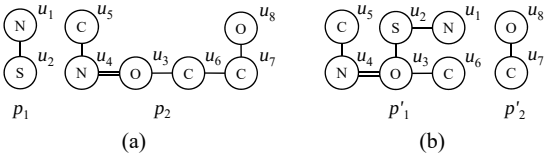


Figure 3: Two ways to partition x in Figure 1

As shown in the example above, the tightness of $lb(x, y)$ is highly dependent on the way to partition x. However, the graph partitioning problem is in general NP-hard [12, 24] and enumerating every possible partitioning to obtain an optimal partitioning is intractable. In the next section, we introduce a measure for a partitioning to develop a good partitioning technique.

3.2 A Qualitative Measure for a Partitioning

Consider a pair of graph x and y with a partitioning $\mathcal{P}(x)$. An inherent limitation of partition-based approaches is that the containment test of each partition is independent, and thus multiple partitions of x can be matching with y in overlapping areas of y. This limitation makes the partition-based bound loose. However, it is hard to tackle the problem because $lb(x, y)$ can exceed the GED if we use a non-overlapping alignment of partitions, where a mismatching partition p is allowed to be aligned to a subgraph of y whose size is less than the size of p. Finding a legal non-overlapping alignment of partitions (i.e., an alignment that results in a minimum lower bound) is computationally impractical.

Beside this fundamental limitation, the following are major problems that make the partition-based lower bound loose.

- P_1 In partition-based approaches, only one edit error is counted from a mismatching partition. A tighter bound can be calculated as $lb(x, y) = \sum_{p \in \mathcal{P}(x) \wedge p \not\sqsubseteq y} sed(p, y)$, where $sed(p, y)$ denotes the subgraph edit distance [22, 26] between p and y.
- P_2 A substructure of x that causes insertion or deletion errors can be divided into multiple partitions. In this case, those edit errors can be hidden between partitions to make the lower bound loose. They can be detected by enumerating every subgraph of x consisting of adjacent partitions, and investigating the subgraphs through subgraph edit distance computations.
- P_3 Edit errors can be buried in edges connecting different partitions and these errors also make the lower bound loose. To precisely find them, we need to solve the problem of placement of partitions into y.

Due to the complexities of subgraph edit distance and partition alignment, the problems above cannot be efficiently solved. The hardness of the limitation and problems also prevents us from accurately analyzing the tightness of $lb(x, y)$. In fact, there is no given proof on the tightness of existing partition-based bounds [6], and it is hard to measure the tightness of $lb(x, y)$ in a quantitative manner. To the best of our knowledge, the only theoretical analysis on the tightness $lb(x, y)$ is that increasing the number of partitions has more chance to get a tighter bound [12]. However, the analysis is based on an assumption that does not take the problem P_2 into consideration. In this paper, instead of a quantitative measure, we introduce a qualitative measure of goodness of a partitioning as stated in the following claim.

Claim 1. *Given two graphs x and y, a partitioning $\mathcal{P}(x)$ is a good partitioning if every mismatching partition $p \in \mathcal{P}(x)$ meets the following conditions.*

- C_1 *Edit errors in p are indivisible, or edit errors in p cannot be distributed over partitions (indivisibility). Ideally, p is minimal, that is, p loses its edit errors and become a matching partition if any vertex in p is removed (mininality).*
- C_2 *An edit error in an edge connecting p to another partition is captured by p, while preserving the condition C_1.*

The indivisibility constraint in C_1 alleviates the problem P_1 since each partition contains the least number of edit errors it can have. The minimality constraint in C_1 alleviates the problem P_2, because by removing unnecessary vertices that do not contribute to edit errors from a partition, those vertices can be combined with other vertices in another partition and cause edit errors. Claim 1 also has the condition C_2 to alleviate the problem P_3.

Although we develop a qualitative measure for a partitioning, it is hard to make a partitioning that exactly meets the measure because a graph partitioning problem even with a simple condition tends to be intractable [24]. Nonetheless, the measure can be a guideline for producing a partitioning to get a tighter bound. In the following sections, we develop a novel partitioning method based on this measure (Section 3.3 for C_1 and Section 3.4 for C_2).

3.3 Incremental Partitioning

In this section, we present a systematic way to produce mismatching partitions that approximately meet the condition C_1 in Claim 1. We begin with the definition of the incremental partitioning strategy.

Definition 4 (Incremental Partitioning). *Given two graphs x and y, an incremental partitioning of x is to extract mismatching partitions from x as follows. Let $V_x = \{u_1, \ldots, u_n\}$. We move the vertices in V_x one after another into a partition p, which is initially empty, while $p \sqsubseteq y$. Let the last vertex moved from x to p be u_l. We finally move u_{l+1} to p to make $p \not\sqsubseteq y$, and produce $\mathcal{P}(x) = \{p, x \backslash p\}$, where $x \backslash p$ denotes the induced subgraph s of x such that $V_s = V_x - V_p$. We repeat this partitioning strategy with $x \backslash p$ until either $x \backslash p \sqsubseteq y$ or $x \backslash p = \emptyset$.*

A graph partitioning produced by the incremental partitioning strategy in Definition 4 satisfies the following property.

Property 1. *Given a pair of graphs x and y, if x is partitioned into $\mathcal{P}(x) = \{p_1, \ldots, p_{k-1}, p_k\}$ using our incremental partitioning strategy, then $p_1, \ldots, p_{k-1}$ are mismatching with y and the last partition p_k, which can be empty, is matching with y. Therefore, $lb(x, y) = k - 1$.*

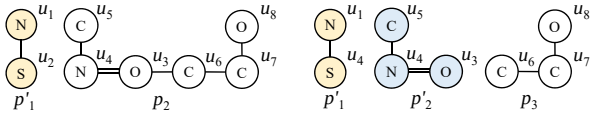
The following lemma states that the incremental partitioning strategy generates a partitioning that exactly meets the indivisibility constraint in the condition C_1 in Claim 1.

Lemma 2. *Given a partitioning $\mathcal{P}(x) = \{p_1, \ldots, p_{k-1}, p_k\}$ produced by the incremental partitioning strategy in Definition 4, it is not possible to divide any partition $p_i \in (\mathcal{P}(x) - \{p_k\})$ into two partitions p_{i1} and p_{i2} such that $p_{i1} \not\sqsubseteq y \ \wedge p_{i2} \not\sqsubseteq y$.*

Proof. For each $p_i = \{u_b, \ldots, u_e\}$ except p_k, our incremental partitioning scheme guarantees that $(p' = p_i - \{u_e\}) \sqsubseteq y$. Since u_e cannot be included in both p_{i1} and p_{i2}, either $p_{i1} \sqsubseteq p' \sqsubseteq y$ or $p_{i2} \sqsubseteq p' \sqsubseteq y$ should be satisfied. □

Example 4. *For a pair of graphs x and y in Figure 1, we incrementally partition x by comparing it with y as follows. Assume that vertices of x are investigated from u_1 to u_8. We first make $\mathcal{P}(x)$ by isolating $\{u_1, u_2\}$ from x into p'_1 as shown in Figure 4(a), because $(p'_1 - \{u_2\}) \sqsubseteq y$ but $p'_1 \not\sqsubseteq y$. Given two partitions of x, we further*

partition p_2 into p'_2 and p_3 by isolating a mismatching partition $\{u_3, u_4, u_5\}$ from p_2, as depicted in Figure 4(b). Since $p_3 \sqsubseteq y$, we cannot proceed the incremental partitioning. Hence, $lb(x, y) = 2$.



(a) $\mathcal{P}(x) = \{p'_1, p_2\}$ (b) $\mathcal{P}(x) = \{p'_1, p'_2, p_3\}$

Figure 4: Incremental partitioning of x in Figure 1

When the incremental partitioning strategy produces a mismatching partition, an induced subgraph isomorphism test is an essential operation. The common principle on subgraph isomorphism test is to visit vertices based on connectivity of vertices and frequencies of vertices and edges [7, 16]. Following the existing solutions, we investigate vertices of x by considering infrequent vertices and edges early while preserving the connectivity.

Given a mismatching partition p in $\mathcal{P}(x)$ generated from the incremental partitioning strategy, we can find an induced subgraph of p that meets the minimality constraint in the condition C_1 as follows. Since the last vertex in p causes the mismatch, we enumerate every induced subgraph of p containing the last vertex and perform an induced subgraph isomorphism test against y to find a subgraph s such that $s \not\sqsubseteq y$ and $|V_s|$ is minimum. This process is obviously time consuming. Instead of finding a minimal one, we propose a method that refines a mismatching partition in $\mathcal{P}(x)$ to approximately meet the minimality constraint.

After we find a mismatching partition p, we rematch p against y using an alternative vertex ordering of p to remove unnecessary vertices from p that do not contribute to edit errors. Let the mismatching partition p be $\{u_1, \ldots, u_f\}$. Because $\{u_1, \ldots, u_{f-1}\}$ is matching with y by Definition 4, u_f causes the mismatching and edit errors are likely to be clustered in u_f and vertices adjacent to u_f. Therefore, by using the vertex u_f as the start vertex and reordering p in the same way (i.e., considering infrequent vertices and edges early while preserving the connectivity), we have a chance to reduce the size of the mismatching partition. The following example illustrates rematching of a mismatching partition to reduce the size of the mismatching partition.

Example 5. *Consider a pair of graphs x and y in Figure 5. Assume the vertices of x is ordered as $\{u_1, u_2, u_3, u_4, u_5, u_6\}$. Based on the order, we isolate $\{u_1, u_2, u_3, u_4\}$ into a separate partition p. In this case, $x \backslash p$ is matching with y and $lb(x, y) = 1$. We reorder vertices in the mismatching partition p into $\{u_4, u_3, u_2, u_1\}$ by using u_4 as the first vertex and preserving the connectivity of the vertices. By rematching p against y using the vertex ordering, we reduce the mismatching partition p to $\{u_4, u_3\}$. From $x \backslash p$, in this case, we can find one more mismatching partition $\{u_1, u_5\}$, which is refined from $\{u_1, u_2, u_5\}$ by the rematching method, and obtain a tighter bound $lb(x, y) = 2$.*

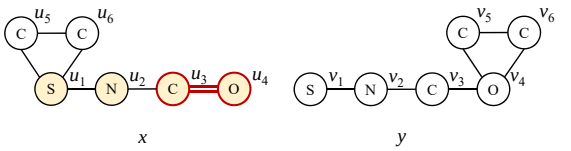


Figure 5: Rematching a mismatching partition

Table 2: Average number of rematching

GED threshold τ	1	2	3	4	5
AIDS	1.41	1.36	1.37	1.38	1.38
PROTEIN	1.50	1.81	1.76	1.69	1.59
PubChem	1.56	1.50	1.47	1.46	1.46

To further reduce the size of a mismatching partition, we repeat rematching while the partition size decreases. As the edit errors are likely to be clustered around the last vertex, we can expect that subgraph isomorphism tests are terminated early and the number of rematching is very small. Table 2 shows the average number of rematching for AIDS, PROTEIN, and PubChem datasets when extracting a mismatching partition (see Section 5 for details of the datasets and queries).

Algorithm 2: IncrementalPartitioning(x, y)

input : x and y are graphs
output: a partition-based GED lower bound $lb(x, y)$

```
1  DetermineVertexOrdering(x);
2  f ← InducedSI(x, y, ∅);
3  if f > |V_x| then return 0 ;
4  p ← first f vertices of x;
5  repeat
6      DetermineRematchingOrdering(p);
7      f ← InducedSI(p, y, ∅);
8      p ← first f vertices of p;
9  until |V_p| does not change;
10 x' ← x\p;
11 foreach connected component c ∈ x' do
12     if |V_c| ≤ α then x' ← x'\c;
13 return 1+IncrementalPartitioning(x', y);
```

Algorithm 2 outlines the incremental partitioning algorithm. Given a pair of graphs x and y, the algorithm computes the lower bound $lb(x, y)$ by partitioning x based on the condition C_1 in Claim 1. It first determines the vertex ordering of x using DetermineVertexOrdering (omitted, Line 1) and then perform an induced subgraph isomorphism test of x against y based on the ordering (Line 2). InducedSI, which will be presented at the end of the next section, identifies and returns the least vertex position in x that makes the matching fail. If the position is greater than the number of vertices in x, then $x \sqsubseteq y$, and return $lb(x, y) = 0$ (Line 3). Otherwise, it extracts the vertices causing the mismatch into a partition p (Line 4).

The algorithm reduces the size of the mismatching partition p using the rematching method (Lines 5–9). DetermineRematchingOrdering (omitted, Line 6) is the same with DetermineVertexOrdering except that it uses the last vertex in p as the start vertex. After reordering vertices in p, the algorithm rematches p against y (Line 7). It repeats rematching while the size of the mismatching partition p shrinks (Line 9). The algorithm finally detaches p from x to make x' (Line 10).

After isolating a mismatching partition p from x, the remaining part of x, which is x', often forms a disconnected graph. We observed that a tiny connected component in a disconnected graph can cause a serious performance problem in subgraph isomorphism test. The existing subgraph isomorphism algorithms

assume connected graphs, and thus they do not pay attention to this problem. To prevent this worst case in subgraph isomorphism test, the algorithm removes each tiny connected component c from x' such that $|V_c| \leq \alpha$, where α is a tunable parameter (Lines 11–12). Then, it recursively identifies the number of mismatching partitions in x' and returns $lb(x, y)$ (Line 13).

Correctness and Complexity of Algorithm 2: Whenever a mismatching partition is identified, the algorithm increments the lower bound by 1 (Line 13). Therefore, the algorithm correctly returns a lower bound by Lemma 1. Assuming the number of rematching is bound to a constant, the worst case complexity is $\sum_{p \in \mathcal{P}(x)} O((\gamma_p \cdot \gamma_p)^{|V_p|}) = O((\gamma_x \cdot \gamma_x)^{|V_x|})$, which is the same as traditional subgraph isomorphism, where γ_g denotes the maximum vertex degree in a graph g.

3.4 Exploiting Bridges

In this section, we propose a novel technique to detect and exploit edit errors buried in those edges connecting different partitions. With the proposed technique, we develop the *bridge constraint* to meet the condition C_2 in our qualitative measure. We first define bridge and then present formulas to count edit errors in bridges.

Definition 5 (Bridge). *Given a partition p, a bridge of a vertex $u \in p$ is an edge connecting u to a vertex $u' \notin p$.*

Lemma 3. *Given a partition p of a graph x and an occurrence o of p in another graph y, suppose a vertex $u \in p$ is mapped to a vertex $v \in o$.*

(1) The number of edit errors between bridges of u and v is

$$\mathcal{B}_e(u, v) = \Gamma(L_{br}(u), L_{br}(v)),$$

where $L_{br}(w)$ denotes the label multiset of the bridges of a vertex w, and $\Gamma(A, B)$ is $\max(|A - B|, |B - A|)$.

(2) The number of edit errors in bridges of p with respect to o is

$$\mathcal{B}(p, o) = \mathcal{B}(M) = \sum_{u \to v \in M} \mathcal{B}_e(u, v),$$

where M denotes the vertex mapping between p and o, which are identified during induced subgraph isomorphism test of p.

Proof. (1) Let $D_1 = L_{br}(u) - L_{br}(v)$ and $D_2 = L_{br}(v) - L_{br}(u)$, and assume $|D_1| \geq |D_2|$. To transform $L_{br}(u)$ to $L_{br}(v)$, we need $|D_2|$ substitutions of labels in D_1 and $|D_1| - |D_2|$ deletions of labels in D_1. That is, we need $|D_2| + |D_1| - |D_2| = |D_1| = \Gamma(L_{br}(u), L_{br}(v))$ edit operations. (2) Since no bridge can shared by multiple vertices in p by Definition 5, the number of edit errors in p is the sum of the number of edit errors in the bridges of p. □

The following example illustrates the number of edit errors in bridges of a matching partition.

Example 6. *In Example 4, consider the matching partition $p_3 = \{u_8, u_7, u_6\}$ and its occurrence $o = \{v_3, v_6, v_7\}$ in y as shown in Figure 6. $\mathcal{B}_e(u_8, v_3) = 3$ because u_8 has no bridge while v_3 has 3 bridges. Likewise, $\mathcal{B}_e(u_7, v_6) = 0$ and $\mathcal{B}_e(u_6, v_7) = 0$. Therefore, $\mathcal{B}(p_3, o) = 3 + 0 + 0 = 3$.*

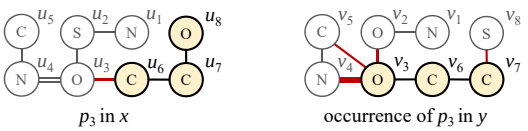


Figure 6: Bridge errors of a matching partition in Figure 4

Given two partitions p and p' of a graph x, suppose that a bridge e connecting p and p' causes one edit error with respect to another graph y. When we count edit errors in bridges of x, the edit error in e is counted twice (i.e., once in p and once in p'). Hence, we can use half of the edit errors counted in bridges so that we do not over-count edit errors in x. Lemma 4 formally states this observation.

Lemma 4. *Given a pair of graphs x and y, consider a matching partition p in x and an occurrence o of p in y. The mapping between p and o causes at least $\lfloor \mathcal{B}(p, o)/2 \rfloor$ edit errors.*

Proof. In this proof, we consider deletion or substitution errors of bridges in x only. Insertion of bridges to x can be proved similarly. Consider we have a partitioning of x such that the i^{th} partition p_i has e_i bridges. Since each bridge is shared by two partitions, bridges should be distributed in a disjoint manner. We distribute bridges to each partition using the following procedure.

```
initially, all bridges are unassigned;
p ← an arbitrary partition;
while there is an unassigned bridge in x do
    if no unassigned bridge is connected to p then
        p ← an arbitrary partition to which at least one
            unassigned bridge connected;
    e ← an unassigned bridge connected to p;
    assign e to p;
    p ← the partition connected to p via e;
```

The procedure above guarantees that at least $\lfloor e_i/2 \rfloor$ bridges are assigned to p_i because if a partition loses a bridge, another bridge (if exists) is always assigned to the partition. If we consider bridges causing edit errors only (i.e., each of e_i bridges causes an edit error), p_i has at least $\lfloor e_i/2 \rfloor$ edit errors. Since there are $\mathcal{B}(p, o)$ edit errors in the bridges connected to p, we can always assign at least $\lfloor \mathcal{B}(p, o)/2 \rfloor$ edit errors to p using the procedure. □

By pushing edit errors in bridges into a matching partition, we can make a rigorous partition matching condition called the bridge constraint as follows.

Corollary 2. *Given a partition p of a graph x and another graph y, p is matching with y if and only if there exists an induced subgraph o of y such that $V_o \subseteq V_y$, $o \sqsubseteq p$, $p \sqsubseteq o$, and $\mathcal{B}(p, o) < 2$.*

Example 7. *In Example 6, since o is the only occurrence of p_3 in y and $\mathcal{B}(p_3, o) \geq 2$, p_3 is mismatching with y by Corollary 2. Therefore, in Example 4, the graph x is divided into four partitions (three mismatching partitions and one empty partition), and we obtain a tighter lower bound $lb(x, y) = 3$.*

Notice that our bridge constraint detects edit errors much more accurately than the half-edge subgraph isomorphism used in existing techniques [12, 24]. For example, in Example 6 and 7, existing techniques cannot detect any edit errors in p_3 (we omit the precise comparison in the interest of space; refer to Pars[24] for the details of the half-edge subgraph isomorphism).

By integrating the bridge constraint with the induced subgraph isomorphism test, we can detect a mismatching partition early to approximately preserve the indivisibility and minimality constraints in C_1 of Claim 1. Algorithm 3 encapsulates our induced subgraph isomorphism test with the bridge constraint.

Algorithm 3: InducedSI(x, y, M)

input : x and y are graphs;
M is a mapping vector (initially $\emptyset$).
output: the least position in x where the matching fails.

1 $iteration \leftarrow |M| + 1$;
2 **if** $iteration > |V_x|$ **then** return $iteration$;
3 $u \leftarrow$ the $iteration^{th}$ vertex in V_x;
4 $C \leftarrow \{v \mid v \in y \wedge v \notin M \wedge L_x(u) = L_y(v)\}$;
5 **foreach** $v \in C$ **do**
6 **if** $\forall u' \rightarrow v' \in M\ L_x(u, u') = L_y(v, v')$ **and** $\mathcal{B}(M \cup \{u \rightarrow v\}) < 2$ **then**
7 $depth \leftarrow$ InducedSI(x, y, $M \cup \{u \rightarrow v\}$);
8 **if** $iteration < depth$ **then** $iteration \leftarrow depth$;
9 **if** $iteration > |V_x|$ **then return** $iteration$;
10 **return** $iteration$;

Like most existing subgraph isomorphism techniques, our algorithm also adopts the Ullmann's algorithm [18] with a difference that ours returns the least vertex position in a partition where the induced subgraph isomorphism test fails. Given a pair of graphs x and y, the algorithm maps the vertices in x one by one to find a mapping M between x and y. For the current vertex u of x (Line 3), it enumerates all unused vertices $v \in y$ whose label is equivalent to the label of u (Line 4), and test if the vertex mapping $u \rightarrow v$ is valid (Lines 6). Then, the bridge constraint in Corollary 2 is applied to the vertex mapping $M \cup \{u \rightarrow v\}$ (Line 6). If it is a valid mapping, the algorithm goes down to the next vertex of x (Line 7). It keeps track of the least position (or maximum iteration count) in x where the induced subgraph isomorphism will fail (Lines 1, 8), and returns the position if $x \not\sqsubseteq y$ (Line 10). If $x \sqsubseteq y$, the algorithm returns $|V_x| + 1$ (Lines 2, 9).

Example 8. *Given a pair of graph x and y depicted in Figure 7, consider we perform* InducedSI$(x, y, \emptyset)$. *Let us assume the vertex ordering of x is from u_1 to u_6. At the first iteration,* InducedSI *adds $u_1 \rightarrow v_7$ into M, and considers $u_2 \rightarrow v_2$ at the second iteration. Because $L_x(u_2, u_1) = L_y(v_2, v_7)$ and $\mathcal{B}(\{u_1 \rightarrow v_7\} \cup \{u_2 \rightarrow v_2\}) = 1$, it adds $u_2 \rightarrow v_2$ into M. At the third iteration, it maps the next vertex u_3 to v_3, and checks the inducedness: $L_x(u_3, u_1) = L_y(v_3, v_7) = \lambda$ and $L_x(u_3, u_2) = L_y(v_3, v_2)$. Then, it tests the bridge constraint and fails to find an occurrence because $\mathcal{B}(\{u_1 \rightarrow v_7, u_2 \rightarrow v_2\} \cup \{u_3 \rightarrow v_3\}) = 2$. Therefore, it returns its iteration count 3, which denotes $\{u_1, u_2, u_3\}$ is a mismatching with y.*

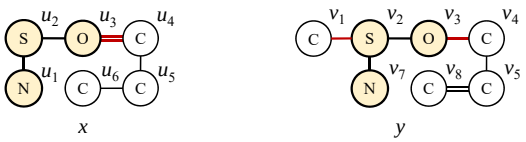


Figure 7: Example of InducedSI

Correctness of Algorithm 3: Given two vertices u and v in a graph g, $L_g(u, v)$ returns a unique value λ when there is no edge between u and v. Therefore, it correctly checks the inducedness of x in Line 6. Mismatching caused by bridge differences is also detected from Line 6, where the correctness is guaranteed by Corollary 2. Because the algorithm basically follows Ullmann's algorithm except the test of inducedness, it correctly computes the induced containment of x. It can be inductively verified the algorithm correctly returns the least position where the isomorphism test fails.

3.5 Verification Algorithm

In this section, we provide Inves verification algorithm. Given a pair of graph x and y and a GED threshold τ, Inves incrementally partitions x to obtain a GED lower bound, and prune the pair if the lower bound is greater than τ. Otherwise, Inves directly calculates the GED between x and y.

Algorithm 4: InvesVerifier(x, y, τ)

input : x and y are graphs; τ is a GED threshold.
output: a boolean value of $ged(x, y) \leq \tau$

1 **if** $\Gamma(L_V(x), L_V(y)) + \Gamma(L_E(x), L_E(y)) > \tau$ **then**
2 **return** false;
3 $lb \leftarrow$ IncrementalPartitioning(x, y);
4 **if** $lb > \tau$ **then return** false;
5 $p \leftarrow$ the last partition of $\mathcal{P}(x)$;
6 **if** $|V_p|/|V_x| > \beta$ **then**
7 $M \leftarrow$ vertex mapping between V_p and V_y;
8 **if** GEDPartial(M, x, y, τ) $\leq \tau$ **then return** true;
9 **return** GED(x, y, τ) $\leq \tau$;

Algorithm 4 shows the details of Inves verification algorithm. Using the label differences of vertices and edges, it first computes a loose GED lower bound and prune the pair if the bound is greater than τ, where $L_V(g)$ and $L_E(g)$ denote the label multisets of vertices and edges in a graph g respectively (Lines 1–2). This technique is originated from the letter-count filter in the problem of DNA read mapping [1, 4] and exploited recently in graph similarity search as a name of the global label filter [25]. Because the global label filter is very simple and highly selective, it is essentially used in graph similarity search (e.g., [24, 25]). After applying the global label filter, Algorithm 4 uses IncrementalPartitioning presented in Algorithm 2 to obtain a partition-based lower bound (Line 3). If the lower bound is greater than τ, it prune the pair (Line 4). We remark that it is obviously optimized by pushing the threshold into IncrementalPartitioning and terminating the partitioning process as soon as $\tau + 1$ mismatching partitions are found.

If the algorithm fails to prune the pair, the last partition $p \in \mathcal{P}(x)$, which can be an empty partition, is matching with y according to Property 1 (Line 5), and a vertex mapping M between p and y is obtained from InducedSI (Line 7). The algorithm exploits this mapping to compute the GED by using it as the initial state of the A* algorithm (i.e., pushing the mapping into the queue instead of an empty mapping in Line 2 of Algorithm 1) (GEDPartial, Line 8). This procedure is called a *partial GED computation*. Notice that the distance calculated by the partial GED computation is an upper bound of the GED of the pair. If it finds the pair meets τ through the partial GED computation, therefore, it can save the time for traversing vertices in M. To prevent frequent invocations of partial GED computation for false positives, we use the partial GED computation only when the size of matching partition is big enough (the tunable parameter β in Line 6). If it

fails to identify if $ged(x, y) \leq \tau$ from the partial GED computation, it finally performs a full GED computation between x and y (Line 9).

Correctness of Algorithm 4: It can be seen GEDPartial correctly returns a GED upper bound. Hence, the correctness of the algorithm is guaranteed by Lemma 1 and Corollary 1.

4 INVES: EFFICIENT GED COMPUTATION

In this section, we develop new methods on top of Algorithm 1 to improve the performance of GED computation. We first propose a method to accurately calculate an estimated distance of a vertex mapping. We then propose a vertex ordering technique that takes advantage of the partitioning results of InvesVerifier.

The performance of the A* algorithm in Algorithm 1 depends on the accuracy of an estimated distance of unmapped vertices and edges. Riesen et al. proposed a bipartite heuristic [14], which gives a lower bound of the distance between unmapped parts with bipartite matching. GSimSearch [25, 26] show that the lower bound of the bipartite heuristic is exactly the same as the label difference in unmapped parts in the unweighted case. This approach does not improve the accuracy of the estimation, but it is significantly faster than the bipartite heuristic because the bipartite heuristic uses the Hungarian algorithm [13] with a high complexity of $O(n^3)$.

To improve the accuracy of the estimated distance, in this paper we distinguish bridges of mapped vertices (i.e., edges connecting mapped vertices to unmapped vertices) from unmapped edges. For a vertex mapping M, each $u \to v \in M$ has $\mathcal{B}_e(u, v)$ edit errors in bridges. Since two different mapped vertices in a graph cannot share any bridges, the total edit errors in the bridges of M are $\mathcal{B}(M)$. Therefore, the estimated distance of M, denoted by $h(M)$, can be calculated by the sum of $\mathcal{B}(M)$ and the label difference in unmapped vertices and unmapped edges except bridges. Formally:

$$h(M) = \mathcal{B}(M) + \Gamma(L_V(x'), L_V(y')) + \Gamma(L_E(x'), L_E(y')),$$

where x' and y' respectively denote the unmapped part of x and y except bridges. The following example illustrates that our method accurately calculates an estimated distance.

Example 9. For the graphs x and y in Figure 8, consider a vertex mapping $\{u_1 \to v_1, u_2 \to v_2, u_3 \to v_3, u_4 \to v_4\}$ is given. The existing distance in the mapping is 1 (due to the label difference between u_2 and v_2). The estimated distance can be calculated in the following two different ways, where the first is the technique used in the previous work while the second is ours.

(1) If we use the label difference of unmapped parts of x and y, we calculate an estimated distance 1 because x has one more single bond (u_5, u_7) in its unmapped part.
(2) If we use the bridge method, the number of edit errors in the bridges is 2 because $\mathcal{B}_e(u_2, v_2) = 1$ and $\mathcal{B}_e(u_4, v_4) = 1$. By using the label difference of the unmapped vertices and the unmapped edges except the bridges, we get an additional edit error 1. By adding the two distances from the bridge method and label filtering, the estimated distance becomes 3.

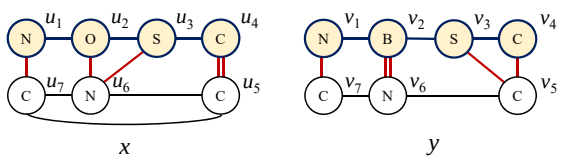


Figure 8: Estimating distance using bridges

The second method is to reorder vertices of the graph x (Line 1 in Algorithm 1). Similar to the problem of subgraph isomorphism, a proper vertex order is also crucial in the GED computation. Since most candidates generated from the filtering phase are false positives, we limit our discussion here to a false positive only (i.e., $ged(x, y) > \tau$). In general, as a vertex mapping M contains more edit errors, the search space of A* algorithm is reduced. For example, consider all edit errors in M and there is no edit error in the remaining vertices and edges. In this case, the A* algorithm can abandon M immediately. To make M contain as many edit errors as possible, therefore, we first consider vertices and edges causing edit errors by placing vertices in the mismatching partitions in the front positions. Since our partitioning method makes the size of a mismatching partition as small as possible, the A* algorithm accurately identifies many edit errors at higher levels of the state-space tree. It is worth noting that our first method is essential to detecting edit errors in those mismatching partitions isolated due to edit errors in bridges.

Algorithm 5: GEDVertexOrder($\mathcal{P}(x)$)

input : $\mathcal{P}(x)$ is the partitioning result of x.
output: O is an ordered set of vertices in x

1 $O \leftarrow \emptyset$;
2 **foreach** *mismatch partition* $p \in \mathcal{P}(x)$ **do** $O \leftarrow O \cup V_p$;
3 DetermineVertexOrdering(O);
4 $O \leftarrow O \cup V_x \backslash O$;
5 Return O;

Another consideration is the connectivity among vertices traversed by the A* algorithm. To reduce the search space, it is important to select the next vertex (Line 8 of Algorithm 1) that is connected to a vertex in M. Algorithm 5 is the vertex ordering algorithm. It first pushes vertices in mismatching partitions to O (Lines 1–2). Then, it orders the vertices in O using DetermineVertexOrdering, which traverses the vertices as described in Section 3 (Line 3). It finally places the remaining vertices (i.e., vertices in a matching partition) at the end (Line 4).

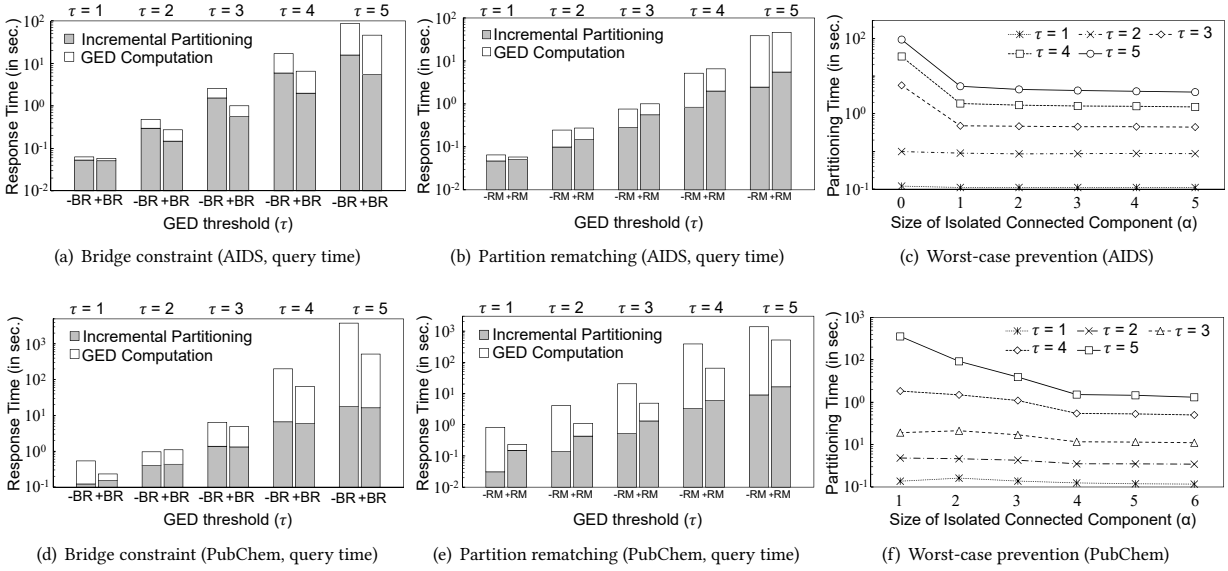
5 EXPERIMENTS

5.1 Experimental Setup

We used the following public real datasets.

- AIDS is an antiviral screen compound dataset containing 42,687 chemical compounds, published by National Cancer Institute[2]. The dataset contains graphs with large size variation. It is a popular benchmark used in many graph search techniques.
- PROTEIN is a protein dataset from the Protein Data Bank[3], containing 600 protein structures. It contains denser and less label-informative graphs.
- PubChem is a chemical compound dataset from the PubChem Project[4]. We used a subset of PubChem consisting of 22,794 chemical compounds. Graphs in the PubChem dataset contain repeating substructures and have less size variation compared with the AIDS and PROTEIN datasets.

[2] http://dtp.nci.nih.gov/docs/aids/aids_data.html
[3] http://www.iam.unibe.ch/fki/databases/iam-graph-database/download-the-iam-graph-database
[4] http://pubchem.ncbi.nlm.nih.gov, Compound_000975001_001000000.sdf



(a) Bridge constraint (AIDS, query time) (b) Partition rematching (AIDS, query time) (c) Worst-case prevention (AIDS)

(d) Bridge constraint (PubChem, query time) (e) Partition rematching (PubChem, query time) (f) Worst-case prevention (PubChem)

Figure 9: Evaluating optimization methods in Inves incremental partitioning

Table 3 summarizes the datasets, where N_{L_v} and N_{L_e} denote the number of distinct vertex and edge labels in the dataset, respectively.

Table 3: Statistics of datasets

Dataset	$\|\mathcal{D}\|$	$\|V\|_{\text{avg}}$	$\|E\|_{\text{avg}}$	$\|V\|_{\max}$	$\|E\|_{\max}$	N_{L_v}	N_{L_e}
AIDS	42,687	25.60	27.60	222	247	62	3
PROTEIN	600	32.63	62.14	126	149	3	5
PubChem	22,794	48.11	50.56	88	92	10	3

For the scalability test, we also used synthetic datasets (see Section 5.3). For each dataset, we conducted experiments using a workload of 100 queries graphs randomly sampled from the dataset[5]. Candidates requiring GED computation, query response time, incremental partitioning time, and GED computation time are measured and reported on the basis of these 100 queries.

All experiments were run on a machine with 32GB RAM, and an Intel core i7 at 3.4 GHz, running a 64-bit Ubuntu OS. We implemented Inves in C++, and compiled it using GCC 4.4.3 with the `-O3` flag[6]. By scanning each dataset once, we pre-computed vertex and edge frequencies, label multisets of vertices and edges of each graph, and the label multiset of edges connected to each vertex. This pre-computation was performed offline and excluded from the query time.

5.2 Evaluating Optimization Methods in Inves

We first evaluated various optimization methods used in Inves. Since the technique is orthogonal to how candidates are generated, we scanned each dataset, and directly applied InvesVerifier on each graph to measure the query time. Notice that y-axis is log-scaled in all experiments.

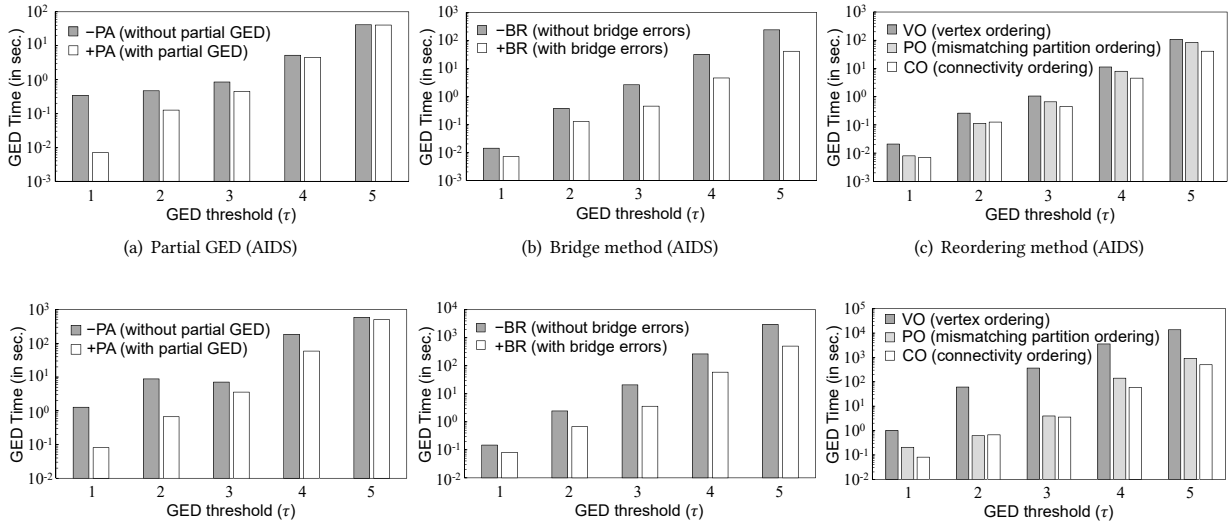
[5]For the PubChem dataset, we manually replaced a few queries because existing techniques did not evaluate those queries in a reasonable amount of time.
[6]The source code of Inves is available at https://github.com/JongikKim/Inves

5.2.1 Methods in Incremental Partitioning. Figure 9(a) and (d) show the effect of the bridge constraint on the AIDS and PubChem datasets. -BR is the InducedSI without the bridge constraint in Corollary 2, and +BR denotes the case where the bridge constraint is added to InducedSI. +BR significantly improved the performance by utilizing differences of bridges connecting different partitions. For example, GED computation time was reduced by 2.4 times on the AIDS dataset and 3.3 times on the PubChem dataset when $\tau = 4$. The significant improvement can be explained by the number of candidates requiring GED computation. Table 4 shows the results on the AIDS and PubChem datasets. As shown in the table, +BR substantially reduced the size of the candidates requiring GED computation.

Table 4: Bridge constraint (number of candidates)

GED threshold (τ)		1	2	3	4	5
AIDS	−BR	135	517	4,686	21,780	60,443
	+BR	125	253	1,086	6,902	30,565
PubChem	−BR	226	397	1,138	5,827	25,354
	+BR	219	369	838	3,675	16,907

Figure 9(b) and (e) show the effect of the partition rematching method on the AIDS and PubChem datasets, where +RM and -RM denotes the results with and without the partition rematching method, respectively. On the PubChem dataset,+RM significantly reduced the GED computation time. For example, the GED time of +RM was about 10, 6, 5, 6, and 3 times faster than -RM for $\tau \in [1, 5]$, respectively. Table 5 shows the number of candidates requiring GED computation with and without the rematching method. Interestingly, the number of candidates reduced by +RM was smaller than that of +BR, while +RM achieved a higher performance gain on GED computation. This is because, although -BR does not use the bridge constraint, the bridge errors are considered in our GED algorithm and those false positives having bridge errors are quickly pruned when computing GED. It can also be explained by the size of mismatching partitions. Since the rematching method reduces the size of mismatching partitions,



(a) Partial GED (AIDS) (b) Bridge method (AIDS) (c) Reordering method (AIDS)

(d) Partial GED (PubChem) (e) Bridge method (PubChem) (f) Reordering method (PubChem)

Figure 10: Evaluating optimization methods in Inves GED computation

A* algorithm can identify more edit errors at higher levels of the state-space tree and prune false positives early. Table 6 shows average sizes of mismatching partitions on the PubChem dataset.

Table 5: Partition rematching (number of candidates)

GED threshold (τ)		1	2	3	4	5
AIDS	−RM	129	279	1,451	8,807	36,265
	+RM	125	253	1,086	6,902	30,565
PubChem	−RM	223	384	951	4,184	18,322
	+RM	219	369	838	3,675	16,907

Table 6: Average mismatching partition size (PubChem)

GED threshold (τ)	1	2	3	4	5
−RM	4.49	17.93	27.34	33.79	37.73
+RM	1.78	9.13	18.28	26.5	31.69

On the AIDS dataset, however, +RM slightly degraded the overall performance because the GED computation on the AIDS dataset is much faster than that on the PubChem dataset and the rematching overhead of +RM is greater than the performance gain on GED computation. From the experiments, we observed that the rematching method should be used for steady performance even though it increases the partitioning time.

Figure 9(c) and (f) show the effect of the tunable parameter α described in Algorithm 2 in Section 3.3. As shown in the figure, tiny connected components greatly degraded the partitioning performance. Without the worst-case prevention method, the partitioning was extremely slow on the PubChem dataset and it did not finish in a reasonable amount of time. For this reason, we omit the result for $\alpha = 0$ in Figure 9(f). Based on the experiment, we used $\alpha = 1$ for the AIDS dataset and $\alpha = 4$ for the PubChem dataset. We performed similar experiments on the PROTEIN dataset, and chose $\alpha = 1$.

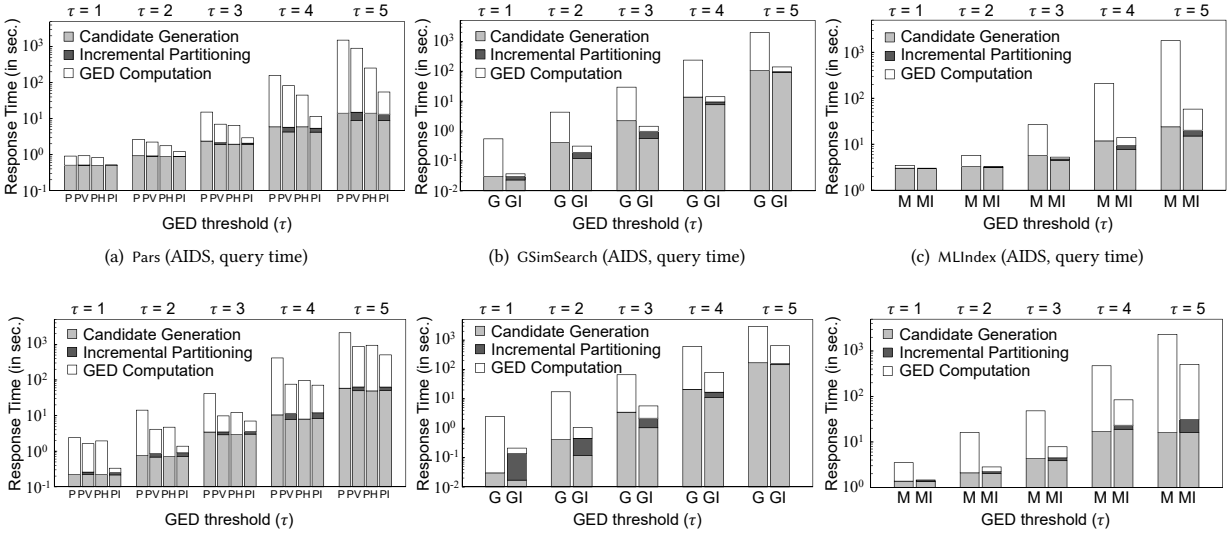
5.2.2 Methods in GED Computation. Evaluation results of the GED computation methods on the AIDS and PubChem datasets are shown in Figure 10. Figure 10(a) and (d) show the effect of the partial GED computation, where -PA and +PA denote InvesVerifier with and without this method, respectively. In our experiments, we observed that the tunable parameter β in InvesVerifier is relatively insensitive, and used $\beta = 0.7$ (we omit the results in the interest of space). +PA showed a good performance for low GED thresholds on both datasets. When a GED threshold was low, most candidates were answers, thus we had more chances to find an answer through a partial GED computation. As the threshold increased, however, most query time was spent on verifying false positives, and this method provided a marginal benefit only.

Figure 10(b) and (e) show the experimental results of our GED computation method with and without exploiting bridge errors, which are denoted by +BR and -BR, respectively. By precisely calculating edit errors in bridges, +BR reduced GED computation time up to 7 times on both datasets.

Figure 10(c) and (f) show the evaluation results of alternative vertex orderings. VO, PO, and CO denote the original vertex order, the vertex order considering vertices in mismatching partitions first, and the vertex order considering connectivity of vertices in Algorithm 5, respectively. PO significantly outperformed VO on both datasets. For example, PO is about 1.5 to 3 times faster than CO on the AIDS dataset. The performance on the PubChem dataset was extremely poor when VO was used. For example, VO was about 100 times slower than PO when $\tau = 3$. This is because the dataset contains graphs having repeating substructures, and thus the A* algorithm cannot efficiently prune the state-space tree without a proper vertex ordering. Considering the connectivity of vertices in different partitions, CO exhibited best performance for all the GED thresholds on both datasets.

5.3 Improving existing techniques

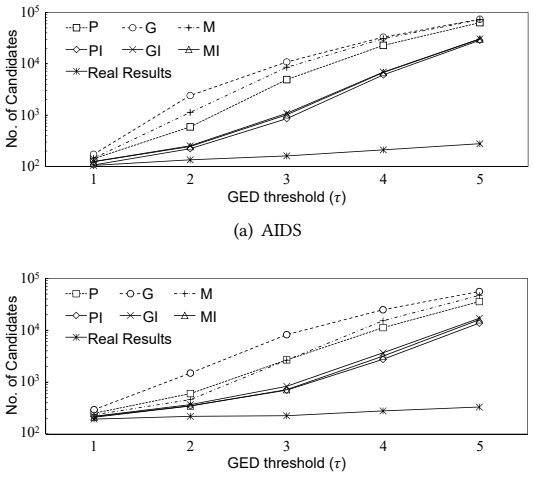
In this experiment, we evaluated how Inves can be adopted by existing techniques to improve their performance. We chose three representative techniques:



(a) Pars (AIDS, query time) (b) GSimSearch (AIDS, query time) (c) MLIndex (AIDS, query time)

(d) Pars (PubChem, query time) (e) GSimSearch (PubChem, query time) (f) MLIndex (PubChem, query time)

Figure 11: Improvement of existing techniques



(a) AIDS

(b) PubChem

Figure 12: Number of candidates

(1) GSimSearch, labeled as G in figures, is a path-based q-gram approach proposed in [25, 26]. According to the results of [25], we used $q = 4$ for the AIDS dataset and $q = 3$ for the PROTEIN dataset. We conducted experiments on gram lengths for the PubChem dataset, and used $q = 4$ based on the results.

(2) Pars, labeled as P, is the state-of-the art partition-based approach [24]. For best performance, we improved Pars in the following ways. Since sequential scanning of indexed partitions was very slow, we implemented an index access method by modifying the SwiftIndex [16]. After we generated candidates using the index access method, we applied their recycling subgraph isomorphism test to each candidate. We also improved its verification process by using the local label filtering and the vertex ordering based on mismatching q-grams proposed in GSimSearch.

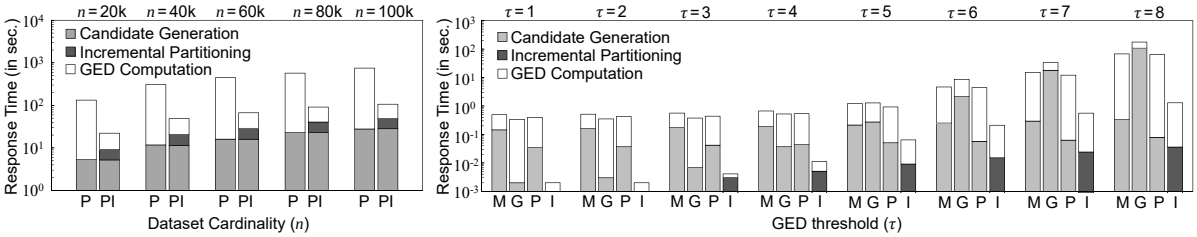
(3) MLIndex, labeled as M, is a multi-layered index technique proposed in [12]. We also improved MLIndex in the same way as Pars.

For each of the three implemented techniques, we adopted Inves in their verification phase. We labeled the corresponding improved techniques as PI, GI, and MI, respectively. We did not include other existing techniques such as c-star [22], SEGOS [20] and k-AT [19] since GSimSearch and Pars consistently outperformed these techniques [24–26]. We also excluded Mixed [27] because its performance results showed no significant differences with the performance of GSimSearch as reported in [3].

Figure 11 shows the results on the AIDS and PubChem datasets (see Figure 13(b) for the results of the PROTEIN dataset). In Figure 11(a) and (d), PV and PH denote Pars with the proposed verifier (without our GED computation methods), and Pars with the bridge method for GED (without the incremental partitioning), respectively. As shown in Figures 11, Inves improved the performance of all these three existing techniques by up to an order of magnitude. The significant improvement can be explained by the results of PV, PH, and PI in Figure 11(a) and (d). Both our partitioning and GED computation methods significantly reduced the total search time. When the incremental partitioning and GED computation methods are used together, the overall performance improvement was even more. In Figure 11(a), for example, when $\tau = 4$, PV was about 2 times faster than P, and PH was about 4 times faster than P. PI was about 25 times faster than P. Similar results were also observed on GSimSearch and MLIndex. Another important indicator of the improvement is the number of candidates requiring GED computation. As shown in Figures 12, Inves generated much smaller sets of candidates for the AIDS and PubChem datasets.

Scalability tests: We also evaluated the scalability of the proposed technique, on both the PROTEIN dataset and synthetic datasets generated by a graph generator[7]. The generator measured the graph size in terms of the number of edges ($|E|$), and

[7] GraphGen (http://www.cse.ust.hk/graphgen/) is a popular graph generator widely used in related work (e.g., [12, 24, 26]) for scalability tests.



(a) Dataset cardinality (Synthetic, $\tau = 3$)

(b) GED threshold (PROTEIN)

Figure 13: Evaluating scalability

the density of a graph defined as $d = 2|E|/|V|(|V| - 1)$. We used $|E| = 20$ and $d = 0.3$ for the experiments, which were default values of the generator. The cardinality of vertex and edge label domains were set to 2 and 1, respectively. Figure 13(a) shows the results. To evaluate the scalability, we generated five synthetic datasets consisting of 20k, 40k, 60k, 80k, and 100k graphs. In the experiments, 100 query graphs were randomly sampled from each dataset and the GED threshold was fixed as 3. The query time grew steadily and the growth ratios of query times were similar in P and PI.

Figure 13(b) shows the scalability results of the GED threshold. Because the PROTEIN dataset contains dense graphs, we chose the dataset to increase τ up to 8. Since the dataset contains 600 graphs only, we separately ran Inves, denoted by I in the figure, by scanning all the data graphs. As shown in the figure, Inves scaled best in terms of response time and outperformed existing techniques by up to about 65 times.

6 CONCLUSIONS

In this paper, we developed a novel technique called Inves for verifying if the graph edit distance (GED) between two graphs is within a threshold, an important and expensive step in graph similarity search. Its main idea is to judiciously and incrementally partition a candidate graph based on the query graph, and use the results to compute a lower bound of their distance. If a full GED computation is needed, Inves utilizes the collected information, and uses novel methods and an A* algorithm to search in the space of possible vertex mappings between the graphs to compute their GED efficiently. We presented a full specification of the technique, and conducted extensive experiments on both real and synthetic datasets. The results showed that the technique can significantly improve the performance of existing techniques [12, 24–26] by an order of magnitude.

ACKNOWLEDGMENTS

This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2018-2015-0-00378) supervised by the IITP (Institute for Information & communications Technology Promotion). Dong-Hoon Choi was partially supported by ETRI R&D Program ("Development of Big Data Platform for Dual Mode Batch-Query Analytics, 14Z1400") funded by the Government of Korea. Chen Li has been partially supported by NSF award III 1745673.

REFERENCES

[1] A. Ahmadi, A. Behm, N. Honnalli, C. Li, and X. Xie. 2012. Hobbes: Optimized gram-based methods for efficient read alignment. *Nucleic Acids Res.* 40 (2012), e41.
[2] H. Bunke and K. Shearer. 1998. A graph distance metric based on the maximal common subgraph. *Pattern Recogn. Lett.* 19, 3-4 (1998), 255–259.
[3] X. Chen, H. Huo, J. Huan, and J. S. Vitter. 2017. Efficient graph similarity search in external memory. *IEEE Access* 5 (2017), 4551–4560.
[4] A. Döring, D. Weese, T. Rausch, and K. Reinert. 2008. SeqAn an efficient, generic c++ library for sequence analysis. BMC Bioinformatics. *BMC Bioinformatics* 9 (2008), 11.
[5] A. Fischer, C. Y. Suen, V. Frinken, K. Riesen, and H. Bunke. 2015. Approximation of graph edit distance based on Hausdorff matching. *Pattern Recognition* 48, 2 (2015), 331 – 343.
[6] K. Gouda and M. Arafa. 2015. An improved global lower bound for graph edit similarity search. *Pattern Recogn. Lett.* 58 (2015), 8–14.
[7] W.-S. Han, J. Lee, and J.-H. Lee. 2013. Turbo$_{\text{ISO}}$: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *SIGMOD Conference*. 337–348.
[8] J. Kim. 2015. An effective candidate generation method for improving performance of edit similarity query processing. *Information Systems* 41 (2015), 116–128. Issue 1.
[9] J. Kim, C. Li, and X. Xie. 2014. Improving read mapping using additional prefix grams. *BMC Bioinformatics* 15 (2014), 42.
[10] J. Kim, C. Li, and X. Xie. 2016. Hobbes3: Dynamic generation of variable-length signatures for efficient approximate subsequence mappings. In *ICDE*. 169–180.
[11] G. Li, D. Deng, J. Wang, and J. Feng. 2011. Pass-Join: A Partition based method for similarity joins. *PVLDB* 5, 3 (2011), 253–264.
[12] Y. Liang and P. Zhao. 2017. Similarity search in graph databases: a multi-layered indexing approach. In *ICDE*.
[13] J. Munkres. 1957. Algorithms for the assignment and transportation problems. *J. SIAM* 5 (1957), 32–38.
[14] K. Riesen, S. Fankhauser, and H. Bunke. 2007. Speeding up graph edit distance computation with a bipartite heuristic. In *MLG*.
[15] H. Shang, X. Lin, Y. Zhang, J. X. Yu, and W. Wang. 2010. Connected substructure similarity search. In *SIGMOD Conference*. 903–914.
[16] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB* 1, 1 (2008), 364–375.
[17] Y. Tian, R. C. Mceachin, C. Santos, D. J. States, and J. M. Patel. 2007. SAGA: A subgraph matching tool for biological graphs. *Bioinformatics* 23, 2 (2007), 232–239.
[18] J. R. Ullmann. 1976. An algorithm for subgraph isomorphism. *J. ACM* 23, 1 (1976), 31–42.
[19] G. Wang, B. Wang, X. Yang, and G. Yu. 2012. Efficiently indexing large sparse graphs for similarity search. *IEEE Trans. on Knowl. and Data Eng.* 24, 3 (2012), 440–451.
[20] X. Wang, X. Ding, A. K. H. Tung, S. Ying, and H. Jin. 2012. An efficient graph indexing method. In *ICDE*. 210–221.
[21] C. Xiao, W. Wang, and X. Lin. 2008. Ed-Join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB* 1, 1 (2008), 933–944.
[22] Z. Zeng, A. K. H. Tung, J. Wang, J. Feng, and L. Zhou. 2009. Comparing stars: On approximating graph edit distance. *PVLDB* 2, 1 (2009), 25–36.
[23] S. Zhang, J. Yang, and W. Jin. 2010. SAPPER: Subgraph indexing and approximate matching in large graphs. *PVLDB* 3, 1 (2010), 1185–1194.
[24] X. Zhao, C. Xiao, X. Lin, Q. Liu, and W. Zhang. 2013. A partition-based approach to structure similarity search. *PVLDB* 7, 3 (2013), 169–180.
[25] X. Zhao, C. Xiao, X. Lin, and W. Wang. 2012. Efficient graph similarity join with edit distance constraints. In *ICDE*. 834–845.
[26] X. Zhao, C. Xiao, X. Lin, W. Wang, and Y. Ishikawa. 2013. Efficient processing of graph similarity queries with edit distance constraints. *The VLDB Journal* 22, 6 (2013), 727–752.
[27] W. Zheng, L. Zou, X. Lian, D. Wang, and D. Zhao. 2015. Efficient graph similarity search over large graph databases. *IEEE Trans. on Knowl. and Data Eng.* 27, 4 (2015), 964–978.
[28] G. Zhu, X. Lin, K. Zhu, W. Zhang, and J. X. Yu. 2012. TreeSpan: Efficiently computing similarity all-matching. In *SIGMOD Conference*. 529–540.