The VLDB Journal (2013) 22:727–752
DOI 10.1007/s00778-013-0306-1

REGULAR PAPER

Efficient processing of graph similarity queries with edit distance constraints

Xiang Zhao · Chuan Xiao · Xuemin Lin · Wei Wang · Yoshiharu Ishikawa

Received: 13 June 2012 / Revised: 12 January 2013 / Accepted: 17 January 2013 / Published online: 20 February 2013

Abstract Graphs are widely used to model complicated data semantics in many applications in bioinformatics, chemistry, social networks, pattern recognition, etc. A recent trend is to tolerate noise arising from various sources such as erroneous data entries and find similarity matches. In this paper, we study graph similarity queries with edit distance constraints. Inspired by the q-gram idea for string similarity problems, our solution extracts paths from graphs as features for indexing. We establish a lower bound of common features to generate candidates. Efficient algorithms are proposed to handle three types of graph similarity queries by exploiting both matching and mismatching features as well as degree information to improve the filtering and verification on candidates. We demonstrate the proposed algorithms significantly outperform existing approaches with extensive experiments on real and synthetic datasets.

Electronic supplementary material The online version of this article (doi:10.1007/s00778-013-0306-1) contains supplementary material, which is available to authorized users.

X. Zhao (✉) · X. Lin · W. Wang
The University of New South Wales, Sydney, Australia
e-mail: xzhao@cse.unsw.edu.au

X. Lin
e-mail: lxue@cse.unsw.edu.au

W. Wang
e-mail: weiw@cse.unsw.edu.au

X. Zhao
NICTA, Sydney, Australia

C. Xiao · Y. Ishikawa
Nagoya University, Nagoya, Japan
e-mail: chuanx@nagoya-u.jp

Y. Ishikawa
e-mail: y-ishikawa@nagoya-u.jp

Keywords Graph similarity query · Edit distance · q-Gram

1 Introduction

Graphs have a wide range of applications and have been utilized to model complex data in biological and chemical information systems, multimedia, social networks, etc. There has been considerable interest in many fundamental problems in analyzing graphs. Various algorithms are devised to solve the problems, including graph pattern mining [24,34,43], graph containment search and indexing [5,35,41], etc.

Due to the existence of noise and inconsistency in data, a recent trend is to study similarity matches among graphs [22,23,26,27,31,32,36,38]. This body of work solves the problem of searching for graphs in a database that approximately contain or are contained by a query. Among the various graph similarity measures used in these studies, graph edit distance [3,21] has been widely accepted for representing distances between graphs. Compared with alternative distance or similarity measures, graph edit distance has three advantages: (1) It allows changes in both vertices and edges; (2) it reflects the topological information of graphs; and (3) it is a metric that can be applied to any type of graphs. Due to these elegant properties, graph edit distance has been used in the context of classification and clustering tasks in various application domains [20]. However, the expensive computation of graph edit distance poses serious algorithmic challenges. To tackle the NP-hardness of the problem [38], a few algorithms have been proposed to either convert it to binary linear programming and compute the bounds [13], or seek unbounded suboptimal answers with heuristics [9].

In this paper, we investigate graph similarity queries with graph edit distance constraints and focus on three types of

queries which cover a wide range of searching and data cleaning tasks in graph database applications:

- Graph similarity search: find data graphs whose edit distances to a query are within a threshold.
- Graph similarity join: find pairs of graphs from two datasets such that the pairs' edit distances are within a threshold.
- Subgraph similarity search: find data graphs that contain subgraphs to which the edit distances from a query are within a threshold.

Due to the expensive computation of graph edit distance, the state-of-the-art approaches to graph or subgraph similarity search with edit distance constraints are mainly based on a filter-and-verify scheme, that is, first generate a set of promising candidates that potentially satisfy necessary conditions for the edit distance constraint, and then verify them by edit distance computation. The κ-AT algorithm [27] borrows the q-gram idea from the solution to string similarity problems [10] and defines a q-gram as a tree consisting of a vertex along with all those that can be reached in q hops. A count filtering condition on the minimum required number of common q-grams is established to qualify the candidates that satisfy the edit distance constraint. However, it suffers from the looseness of the lower bound due to the impact of edit operations on common q-grams and therefore is only effective against sparse graphs. The choice of q-gram length is also limited to very small values, but short q-grams usually result in poor selectivity and consequently large candidate size. The star structure [38] is exactly the same feature as the 1-gram defined by κ-AT. Unlike κ-AT, it computes the lower and upper bounds of graph (or subgraph) edit distance through bipartite matching between the star representations of two graphs. For graph similarity search, it has to invoke bipartite matching for the query with every data graph. The time complexity will be $O(|R| \cdot |V|^3)$, where $|R|$ is the dataset size and $|V|$ is the number of vertices in a graph. Thus, an immediate remedy is to take advantage of indexes. To this end, an indexing and query processing framework SEGOS [31] is proposed recently. Based on a two-level index structure, SEGOS adopts a novel search strategy adapted from the threshold-based algorithm (TA) and the combined algorithm (CA) [8] to enhance the star structure-based solution. As a result, SEGOS is superior from both perspectives of indexing and searching strategy. However, implicit parameters hidden in the algorithm need to be tuned in order to achieve good performance. Moreover, graph edit distance computation is not involved in its evaluation, and hence, the overall runtime performance of SEGOS remains unclear.

Distinct from existing approaches, we explore a novel perspective of utilizing *path-based* q-grams. We find that the count filtering condition of path-based q-grams is stricter than that of tree-based q-grams. This enables us to perform similarity queries on denser graphs as well as choose longer q-grams for better selectivity. Another novelty is to exploit the valuable information provided by *mismatching* q-grams that do not match in a candidate pair. Two filtering conditions are accordingly proposed so that the size of the candidate set can be substantially reduced. In addition, we leverage the vertex *degree* information to devise a new q-gram matching condition. We also elaborate how to speed up graph edit distance computation by further utilizing the filtering conditions. As a consequence, three algorithms are designed, respectively, to handle the three types of similarity queries. The superior time efficiency against alternative methods is demonstrated by extensive experimental evaluations.

A preliminary version of this paper appeared in [42]. In this version, we make substantial improvements:

- We devise algorithms for graph and subgraph similarity search queries, which are non-trivial extensions of the GSimJoin algorithm proposed in [42]. For graph similarity join queries, R-S join scenario was not covered by [42], but is discussed in this paper. We also discuss the adaptation to directed multigraphs.
- We devise a novel q-gram matching condition by exploiting the vertex degree information in q-grams. The proposed technique is orthogonal to the two major filtering techniques proposed in [42] and substantially reduces the size of the candidate set from GSimJoin.
- We evaluate the effect of the new techniques and the performance on the three types of graph similarity queries with more experiments.

The rest of the paper is organized as follows: Sect. 2 presents the problem definition and preliminaries. Section 3 introduces the definition of path-based q-gram and the basic algorithmic framework for graph similarity search. Sections 4 and 5 present two filtering techniques exploiting mismatching q-grams. Section 6 advances another idea to leverage degree differences when matching q-grams. Section 7 elaborates the verification of candidates. Extensions to graph similarity join and subgraph similarity search are covered in Sects. 8 and 9, respectively. Further adaptation to directed multigraphs are discussed in Sect. 10. In Sect. 11 are experimental results and analyses. Section 12 summarizes related work, followed by conclusion in Sect. 13.

2 Preliminaries

2.1 Problem definition

For the ease of exposition, we focus on *simple* graphs first and postpone the extension to other graphs in Sect. 10. A

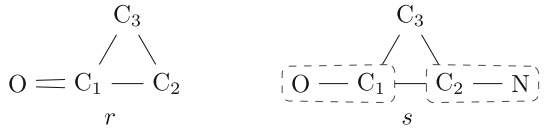

Fig. 1 Cyclopropanone and 2-aminocyclopropanol

simple graph is an undirected graph with neither self-loops nor multiple edges. A labeled graph r can be represented in a quadruple (V, E, l_V, l_E), where V is a set of vertices, $E \subseteq V \times V$ is a set of edges, and l_V (resp. l_E) is a labeling function that assigns labels to vertices (resp. edges). $V(r)$ (resp. $E(r)$) denotes the vertex (resp. edge) set of r. $|V(r)|$ and $|E(r)|$ represent the number of vertices and edges in r, respectively. $l_V(u)$ denotes the label of a vertex u, and $l_E(e(u, v))$ denotes the label of an edge between u and v, $u, v \in V$.

Definition 1 (*graph isomorphism*) A graph r is isomorphic to another graph s if there exists a *bijection* $f : V(r) \to V(s)$ such that (1) $\forall u \in V(r)$, $f(u) \in V(s) \wedge l_V(u) = l_V(f(u))$, and (2) $\forall e(u, v) \in E(r)$, $e(f(u), f(v)) \in E(s) \wedge l_E(e(u, v)) = l_E(e(f(u), f(v)))$.

Definition 2 (*subgraph isomorphism*) A graph r is subgraph isomorphic to another graph s (denoted $r \sqsubseteq s$), if there exists an *injection* $f : V(r) \to V(s)$ such that (1) $\forall u \in V(r)$, $f(u) \in V(s) \wedge l_V(u) = l_V(f(u))$, and (2) $\forall e(u, v) \in E(r)$, $e(f(u), f(v)) \in E(s) \wedge l_E(e(u, v)) = l_E(e(f(u), f(v)))$. r is also called a *subgraph* of s.

A *graph edit operation* is an edit operation to transform one graph to another [3,21]. It can be one of the following six operations:

- insert an isolated labeled vertex into the graph,
- delete an isolated labeled vertex from the graph,
- change the label of a vertex,
- insert a labeled edge into the graph,
- delete a labeled edge from the graph,
- change the label of an edge.

The *graph edit distance* between r and s, denoted by $ged(r, s)$, is the minimum number of edit operations that transform r to a graph isomorphic to s. It is easy to show that graph edit distance is a *metric*. Computing the graph edit distance between two graphs is proved to be NP-hard [38]. For brevity, we use "edit distance" for "graph edit distance" in the rest of the paper when there is no ambiguity.

Example 1 Figure 1 sketches the molecular structures of cyclopropanone (r) and 2-aminocyclopropanol (s) omitting hydrogen atoms. They are used in the investigation of potential antiviral drugs [7]. For ease of illustration, subscripts are added to the carbon atoms, while C_1, C_2, and C_3 correspond to an identical label; single and double lines indicate different chemical bonds, modeled by edge labels in real data. The graph edit distance between r and s $ged(r, s) = 3$, for example, in r inserting an N-labeled vertex and a single edge between C_2 and N, then replacing the double edge with a single edge.

In this paper, we study three types of graph similarity queries based on edit distance, namely graph similarity search, graph similarity join, and subgraph similarity search. The graph similarity search query is formalized as follows.

Problem 1 (*graph similarity search*) Given a data graph collection R, a query graph s, and edit distance threshold τ, graph similarity search is to find all the graphs r from R such that the edit distance between r and s is no larger than τ, that is, $\{ r \mid ged(r, s) \leq \tau, r \in R \}$.

Running multiple graph search queries in a batch mode results in a graph similarity join query.

Problem 2 (*graph similarity join*) Given two graph collections R and S, graph similarity join with edit distance threshold τ returns pairs of graphs from each collection, such that their edit distance is no larger than τ, that is, $\{ \langle r, s \rangle \mid ged(r, s) \leq \tau, r \in R, s \in S \}$. A self-join associates a collection with itself, that is, given a graph collection R, it returns $\{ \langle r_i, r_j \rangle \mid ged(r_i, r_j) \leq \tau \wedge r_i.id < r_j.id, r_i, r_j \in R \}$.

It is also desirable to discover data graphs which approximately contain given queries, and therefore, we consider the subgraph similarity search query.

Problem 3 (*subgraph similarity search*) Given a data graph collection R, a query graph s, and edit distance threshold τ, graph similarity search is to find all the graphs r from R such that there exists a subgraph r' of r to which the edit distance from s is no larger than τ, that is, $\{ r \mid ged(r', s) \leq \tau \wedge r' \sqsubseteq r, r \in R \}$.

Next, we first study graph similarity search and defer the extension to graph similarity join and subgraph similarity search to Sects. 8 and 9. Additionally, we focus on in-memory implementation when describing algorithms.

2.2 Tree-based q-gram approach

A problem related to graph similarity queries is string similarity queries with edit distance constraints, which have been extensively studied since last decade, [15,17,28,29] to name a few recent advances. Among them, several prevalent approaches are based on *q-grams* [17,33], namely substrings of length q. Since an edit operation only affects a limited number of q-grams, similar strings will have certain amount of overlap between their q-gram sets.[1] Based on this

[1] q-Grams in strings are accompanied by their starting positions in the string, and thus there is no duplicate.

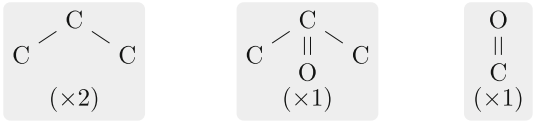

Fig. 2 Tree-based q-grams

observation, these approaches essentially relax the edit distance constraint to a weaker count constraint on the number of common q-grams, called *count filtering*.

Inspired by the idea of q-gram on string similarity queries, [27] proposes κ-AT algorithm that defines q-grams on graphs based on *trees*. For each vertex u, a *tree-based q-gram* is a set of vertices that can be reached from u in q hops, represented in a breadth-first-search tree rooted at u.

Example 2 Consider graph r in Fig. 1, and $q = 1$. r embodies four 1-grams, as shown in Fig. 2, with the first 1-gram appearing twice.

The maximum number of tree-based q-grams that can be affected by an edit operation is shown as $D_{tree} = 1 + \gamma \cdot \frac{(\gamma-1)^q-1}{\gamma-2}$, where γ is the maximum vertex degree in the graph. κ-AT algorithm advises that if graphs r and s are within edit distance τ, they must share at least

$$LB_{tree} = \max(|V(r)| - \tau \cdot D_{tree}(r), |V(s)| - \tau \cdot D_{tree}(s))$$

common q-grams. A pair of graphs conforming to the lower bound is a *candidate* pair. Note it does not necessarily satisfy the edit distance constraint. Hence, edit distance calculation is invoked for every candidate pair.

κ-AT algorithm is observed to have loose lower bound on common q-grams when (1) there is a vertex with high degree in the graph, (2) the distance threshold is large, or (3) q is large. The lower bound can even become less than or equal to zero. We call such phenomenon *underflowing*. This issue results in the following dilemma: We have to use very short q-grams, for example, 1-grams, to ensure the pairs of graphs to have at least one common q-gram so that the all-pair comparison brought about by underflowing can be avoided; however, using short q-grams suffers from poor performance as they are usually frequent and hence yield large candidate set. Considering the two graphs in Fig. 1 and $\tau = 1$, the lower bound is only 1 if we use 1-grams and becomes non-positive under larger distance thresholds or with longer q-grams.

Another approach for graph similarity search [38] is based on star representations of graphs. A star structure is exactly a 1-gram defined by κ-AT; nevertheless, it does not apply the count filtering to approach the problem. With a distinct flavor, it utilizes bipartite matching to derive lower and upper bounds of edit distance for punning and validation, respectively. As a step further, SEGOS [31] enhances star structures with a two-level index. In the upper level, stars from data graphs are used to index the graphs in inverted lists; in the lower-level index, each star is broken into multiple vertices and indexed in inverted lists. The performance of SEGOS is dependent on the parameters that control the access to its two-level index. Edit distance computation is not involved in the experimental evaluation in [31] either, and hence, the overall runtime performance remains unclear. We will compare with SEGOS in the experimental study, equipping it with edit distance computation.

3 A path-based q-gram method

Seeing the drawback of tree-based q-grams, we quest for a new way of defining q-grams on graphs. Akin to q-grams on strings which are essentially sequences, we may choose paths in a graph as its q-grams, which are convertible to sequences. Next, we formally introduce path-based q-grams.

3.1 Definition of path-based q-gram

A *path* in a graph is a sequence of vertices and edges such that there is an edge between any consecutive vertices. A path is *simple* if there are no repeated vertices. The *length* of a path is the number of edges in the path.

Definition 3 (*path-based q-gram*) A path-based q-gram in a graph r is a simple path of length q.

Given a path, we have two label sequences starting from two *terminal vertices*, by sequentially concatenating the labels of its vertices and edges. Nonetheless, we only associate the lexicographically smaller one to a path-based q-gram w as its label sequence, denoted by $seq(w)$, for example, `C-N` and `N-C` are two label sequences of a path-based q-gram w, but we only keep `C-N` as $seq(w)$, as it is lexicographically smaller.

Thus, each path-based q-gram w has a label sequence $seq(w)$ of length $2q+1$. w is *symmetric* if its label sequence is symmetric, for example, `C-O-C`. Since the length of a path can be zero in the case of a single vertex, 0-gram is defined to be a single vertex. The number of paths in a graph is in $O(|V| \cdot \gamma^q)$, where γ is the maximum vertex degree. Compared with tree-based q-grams, decomposing a graph into path-based q-grams increases the total number of q-grams. In the rest of the paper, we use "path-based q-gram" and "q-gram" interchangeably when there is no ambiguity.

Example 3 Consider the two graphs in Fig. 1 and $q = 1$. Assume we take atom symbols as vertex labels and use "–" and "=" as edge labels for single and double bonds, respectively. There are four 1-grams in r:

`C = O` (×1) `C – C` (×3),

and five 1-grams in s:

$\mathtt{C}-\mathtt{N}\ (\times 1) \quad \mathtt{C}-\mathtt{O}\ (\times 1) \quad \mathtt{C}-\mathtt{C}\ (\times 3).$

Given a q-gram size q, we say path-based q-grams w_r and w_s *match* if they correspond to the same label sequence, and they are *matching* q-grams, otherwise *mismatching* q-grams. It is formally stated in Condition 1.

Condition 1 (label-based match) *Consider q-grams w_r and w_s, as well as corresponding label sequences $seq(w_r)$ and $seq(w_s)$, respectively. w_r matches w_s, if $seq(w_r) = seq(w_s)$.*

The time complexity of checking whether two q-grams match, that is, comparing labels one by one, is $O(q)$. We can compare the hash codes of the sequences and hence reduce the time complexity to $O(1)$. This may introduce false positives due to hash collision but no false negatives, and thus the correctness of any filtering algorithms relying on the matching condition is not affected. In practice, most false positives can be avoided by choosing an appropriate hash function. From now on, we assume there is no hash collision and two q-grams match with respect to Condition 1 if the hash codes of their label sequences coincide.

Given graphs r and s, we extract all the paths in the two graphs to make two q-gram sets Q_r and Q_s. Two q-grams $w_r \in Q_r$ and $w_s \in Q_s$ are *common* if w_r matches w_s; equally, we say one q-gram (either w_r or w_s) is *shared* by r and s. We abuse the notation $Q_r \cap Q_s$ to denote the common q-grams between Q_r and Q_s; and $Q_r \backslash Q_s$ denotes the q-grams from Q_r that cannot match any q-gram in Q_s.

Similar to tree-based q-grams, a count filtering condition for path-based q-grams can be developed to relax the edit distance constraint to a weaker count constraint on the number common q-grams. Before presenting that, we first study the effects of edit operations on a graph's q-grams. Let Q_r^u denote the set of q-grams containing vertex u, and Q_r^{uv} denote the set of q-grams containing two consecutive vertices u and v. We say a q-gram is *affected* by an edit operation if the edit operation changes the q-gram's label sequence. Theorem 1 reveals at most how many q-grams in Q_r are affected when an edit operation occurs in r.

Theorem 1 *An edit operation on graph r affects at most $D_{path}(r) = \max_{u \in V(r)} |Q_r^u|$ q-grams in Q_r.*

Proof We enumerate the effects of various edit operations:

- *Insert an isolated labeled vertex into the graph.* No q-grams in Q_r are affected.
- *Delete an isolated labeled vertex from the graph.* The number of q-grams affected is either 1 when $q = 0$, or 0 otherwise.
- *Change the label of a vertex.* Changing vertex u's label affects $|Q_r^u| \leq \max_{u \in V(r)} |Q_r^u|$ q-grams.
- *Insert an labeled edge into the graph.* No q-grams in Q_r are affected.
- *Delete an labeled edge from the graph.* Supposing $e(u, v)$ is deleted, the number of affected q-grams is $|Q_r^{uv}| \leq \max_{u \in V(r)} |Q_r^u|$.
- *Change the label of an edge.* It affects the same number of q-grams as deleting an edge from the graph. □

According to Theorem 1, the count filtering condition for path-based q-grams can be established in Lemma 1.

Lemma 1 (count filtering) *Consider two graphs r and s. If $ged(r, s) \leq \tau$, they must share at least*

$$LB_{path} = \max(|Q_r| - \tau \cdot D_{path}(r), |Q_s| - \tau \cdot D_{path}(s))$$

common q-grams.

Example 4 Consider Fig. 1, $\tau = 1$, and $q = 1$. Changing the label of $\mathtt{C}_1$ gives the maximum $|Q_r^u| = 3$ for both graphs. Hence, the lower bound of common path-based q-grams between r and s is $\max(4-3, 5-3) = 2$. If we increase the q-gram length to 2, the lower bound of path-based q-grams is still above zero, as given by $\max(5-5, 7-6) = 1$, whereas using tree-based q-grams provides a lower bound of -5.

A subtle case in counting common q-grams is: When a q-gram in Q_r matches two q-grams in Q_s, adding up two common q-grams results in *multiple matching*, since only one indeed matches. In general, if mq-grams from Q_r match nq-grams from Q_s such that these $m+nq$-grams correspond to an identical label sequence, at most $\min(m, n)$ q-grams contribute to the common q-grams. Multiple matching is avoided when counting the number of common q-grams and will be further discussed in Sect. 7.1.

3.2 Comparison with tree-based q-grams

Now, we compare the influence of edit operations on tree-based and path-based q-grams.

- For $q = 1$, consider r in Fig. 1. All the tree-based q-grams, which cover the whole graph, can be affected by an edit operation on $\mathtt{C}_1$, while the path-based q-gram consisting of $\mathtt{C}_2$ and $\mathtt{C}_3$ remains unaffected. This example showcases path-based q-grams can preserve more common structural information than tree-based q-grams, excluding the affected part.
- For longer q-grams, the number of vertices covered by a tree-based q-gram increases *exponentially* with q. One edit operation on any of the vertices makes the q-gram mismatch. On the contrary, the coverage of a path-based q-gram increases *linearly* with q, and therefore, the probability of being hit by an edit operation is decreased.

– The number of path-based q-grams grows exponentially with q, given by $O(|V| \cdot \gamma^q)$, while it is $|V|$ for tree-based q-grams. As a consequence, we would have more path-based q-grams left after applying τ edit operations than tree-based q-grams, due to larger total number of q-grams, and lower probability of being hit by edit operations.

Experimental results have suggested that path-based q-grams have the advantage of presenting tighter count filtering lower bounds over tree-based q-grams.[2] This potentially deliver the chance of using longer q-grams in seek of better selectivity and runtime performance. We remark that using path-based q-grams cannot get rid of the underflowing issue in extreme cases; however, it reduces the chance of underflowing, compared with tree-based q-grams.

3.3 Prefix filtering

An efficient way to find the pairs of graphs that satisfy the count filtering condition is to use an inverted index [2]. An inverted index maps each q-gram w to a list of identifiers of the graphs that contain w. With an inverted index built on data graphs, for a query graph, we scan its q-grams and use each of them to probe the inverted list in order to collect data graph identifiers as well as their occurrence numbers. All the graphs whose occurrence numbers meet the count filtering lower bound are candidates of the query.

The main performance bottleneck in accessing inverted index is that the inverted lists of some q-grams can be fairly long, for example, the carbon chain C − C − C exists in most organic compounds. These long inverted lists incur prohibitive accessing overhead, and a large number of candidates will be produced if they share such q-grams with the query. Existing approaches to string similarity problem address it by employing *prefix filtering* [4,17,33] to quickly prune the candidates that are guaranteed to not meet the count filtering condition. The intuition is that if two sets of q-grams meet the lower bound constraint, they must share at least one common q-gram if we look into part of the q-grams.

Figure 3 illustrates the idea of prefix filtering. Suppose q-grams are *sorted* by the same ordering, and l is the number of q-grams in both sets. The unshaded cells are *prefixes*, for example, w_a and w_b are Q_r's prefixes. If Q_r and Q_s have no common q-grams in their prefixes, the number of their common q-grams is no more than $LB_{path} - 1$. We formally state the prefix filtering principle for graph similarity queries.

Lemma 2 (prefix filtering) *Consider graphs r and s, their q-gram sets Q_r and Q_s, sorted by a global ordering $\mathcal{O}$ of the q-gram universe, respectively. Let p-prefix denote the first p elements in a set. If $|Q_r \cap Q_s| \geq \alpha$, the $(|Q_r| - \alpha + 1)$-prefix of Q_r and the $(|Q_s| - \alpha + 1)$-prefix of Q_s must share at least one common q-gram.*

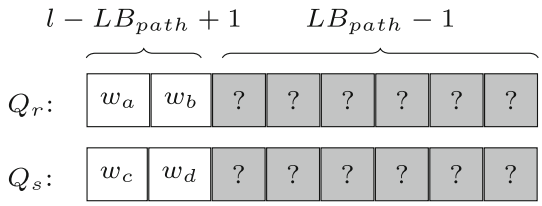

Fig. 3 Illustration of prefix filtering

In order to achieve fewer candidates and faster execution, we sort the q-grams set of each graph in ascending order of *document frequency* of label sequences, that is, the number of graphs containing the q-gram's label sequence. In this way, q-grams with rare label sequences reside ahead of those with frequent ones in q-gram sets. Intuitively, label sequences with low (resp. high) document frequencies are possessed by less (resp. more) graphs. Sorting q-grams in this order is a good heuristic to speed up similarity queries [4].

3.4 Graph similarity search algorithm

Combining count filtering and prefix filtering, we are ready to present the basic GSimSearch algorithm for similarity search queries (Algorithm 1). It consists of two phases: indexing (Algorithm 2) and query answering (Algorithm 3). In the indexing phase, the algorithm takes as input a graph database R and a distance threshold τ, and constructs an inverted index. It first decomposes each data graph into a set of q-grams and indexes its prefix by incorporating prefix filtering (Lines 3–7 in Algorithm 2). In the query answering phase, the algorithm receives a query graph s and decomposes it into a q-gram set, sorted in the same global order as in the indexing phase. For each q-gram w in its prefix, it probes the inverted index to collect the data graphs containing w in their prefixes. The candidates are sent into Verify and checked by (1) count filtering and then (2) the expensive edit distance computation. According to Lemmas 1 and 2, the prefix length is $\tau \cdot D_{path}(r) + 1$ for each r (Line 4 in Algorithm 2), and $\tau \cdot D_{path}(s) + 1$ for query s (Line 4 in Algorithm 3). In addition, the numbers of vertices and edges in r and s must have a difference within τ. This *size filtering* is included in Line 8 in Algorithm 3.

The indexing phase can be done offline if we are offered the storage to keep the indexes. Although the edit distance threshold τ may not be given beforehand in some cases, observing that the prefixes under higher distance thresholds always subsume the prefixes under lower distance thresholds, we can build index for a fixed threshold $\tau_{\max}$ and it can be used for all similarity queries with $\tau \leq \tau_{\max}$.

[2] Rare cases are observed that tree-based q-grams deliver identical or even tighter count filtering lower bound than path-based q-grams.

Algorithm 1: GSimSearch (R, s, τ)

Input : R is a collection of graphs; s is a query graph; τ is an edit distance threshold.
Output : A set of query result T.

```
1 I ← GSimIndex (R, τ) ;            /* build index */
2 T ← GSimQuery (s, I, τ) ;         /* find results */
```

Algorithm 2: GSimIndex (R, τ)

Input : R is a collection of graphs; τ is an edit distance threshold.
Output : An inverted index I built on R.

```
1 I_i ← ∅ (1 ≤ i ≤ |U|) ;           /* inverted index */
2 for each r ∈ R do
3     Q_r ← r's q-grams sorted in 𝒪;
4     p_r ← τ · D_path(r) + 1;
5     for i = 1 to p_r do
6         w ← Q_r[i];
7         I_w ← I_w ∪ {r} ;    /* index for q-gram w */
8 return I
```

Algorithm 3: GSimQuery (s, I, τ)

Input : s is a query graph; I is R's inverted index; τ is an edit distance threshold.
Output : $T = \{r \mid ged(r, s) \leq \tau, r \in R\}$.

```
1  T ← ∅;
2  A ← empty map from id to boolean;
3  Q_s ← s's q-grams sorted in 𝒪;
4  p_s ← τ · D_path(s) + 1;
5  for i = 1 to p_s do
6      w ← Q_s[i];
7      for each r ∈ I_w such that A[r] has not been initialized do
8          if abs(|V(r)| − |V(s)|) + abs(|E(r)| − |E(s)|) ≤ τ
           then
9              A[r] ← true ;      /* find a candidate */
10 T ← T ∪ Verify(s, A);
11 return T
```

This completes the basic algorithmic framework for graph similarity search. In the following sections, first we study how to exploit the information provided by mismatching q-grams to gain efficiency. Although similar property also happens to strings [33], the scenario on graphs is much more challenging: (1) q-grams on strings have starting positions, and hence, are easy to locate, while q-grams on graphs do not have such attribute; and (2) the minimum edit operation problem on strings is of polynomial time complexity while we will show it is NP-hard on graphs. We propose non-trivial techniques for graphs to reduce both index and candidate sizes. Moreover, we present a stricter matching condition for path-based q-gram to further reduce candidates by integrating more structural information, followed by an optimized verification algorithm.

4 Minimum edit filtering

We first show an illustrative example.

Example 5 Consider Fig. 1, $\tau = 1$, and $q = 1$. The count filtering lower bound is 2; the two graphs share 3 q-grams (see Examples 3 and 4), and thus survive the filter. However, the two mismatching q-grams in s—C-O and C-N—are disjoint (bounded regions in s). At least two edit operations are required to affect both of them. We infer an edit distance lower bound between r and s to be 2, and hence prune the pair. This motivates us to find the minimum number of edit operations that cause the observed mismatching q-grams.

The above example evidences the implication of edit operations occurring on mismatching q-grams and the existence of redundancy within prefixes. Both index size and number of candidates passing prefix filtering can be reduced if we are able to shorten prefix lengths.

4.1 Minimum graph edit operations

Example 5 illustrates the case of disjoint mismatching q-grams. To handle the general case of overlapping q-grams, we formulate the *minimum graph edit operation* problem.

Problem 4 (*minimum graph edit operation*) Given a set of q-grams Q, find the minimum number of graph edit operations that can affect all the q-grams in Q.

Theorem 2 *The minimum graph edit operation problem is NP-hard.*

Proof First, we prove that only the operation of changing vertex label needs to be considered. For any vertex edit operation, the affected q-grams are a subset of those affected by changing the vertex's label. For any edge edit operation, the affected q-grams are also subsumed by those affected by changing the vertex label of the edge's either end.

Second, we show a polynomial reduction from the minimum set cover problem. Consider a universe $\mathscr{U}$ of elements and n sets whose union constitutes the universe. Each element is treated as a q-gram, and for each set containing multiple elements, we let these q-grams overlap on a vertex. Therefore, changing the label of this vertex affects these q-grams, that is, covering these elements. Then, the minimum number of edit operations that affect the q-grams in Q is exactly the minimum number of sets whose union covers all elements in $\mathscr{U}$. By reduction from the set cover problem, the minimum graph edit operation problem is NP-hard. □

Despite its NP-hardness, the problem is solvable with an exact algorithm enumerating the positions of τ edit operations, since we only concern whether the answer is within τ.

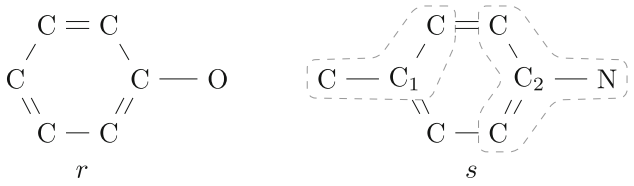

Fig. 4 Example of minimum edit operation

The worst-case time complexity is $O(|V_Q|^\tau + |Q|)$, where V_Q is the set of vertices contained by the q-grams in Q.

One may notice the reduction in the proof of Theorem 2 is a direct problem mapping. It is straightforward to show a minimum edit operation problem can be reduced to a minimum set cover problem via the reverse mapping. Thus, we conclude the minimum set cover problem and minimum edit operation problem are equivalent. As a consequence, to alleviate the problem of large $|V_Q|$, we may compute an approximate answer using the greedy algorithm for the minimum set cover problem, with an approximation ratio of $\ln|Q| - \ln\ln|Q| + 0.78$ [25]. Algorithm 4 encapsulates the approximate algorithm and is guaranteed to return a lower bound of the exact answer in $O(\tau(|V_Q| + |Q|)\log|Q|)$ time.

Example 6 Figure 4 shows the structure of phenol (r) and toluidine (s) molecules. Supposing $q = 2$, there are three mismatching q-grams in s: `C-C-C`, `C-C-N`, and `C=C-N`, as bounded by dashed lines. At least, two edit operations are needed to affect them, for example, changing the vertex labels of C_1 and C_2.

It is noteworthy to mention the following two properties, which are essential to the filtering techniques we are going to present. Let $min\text{-}edit(Q)$ denote the minimum number of graph edit operations on a set of q-grams Q.

Proposition 1 (monotonicity) $min\text{-}edit(Q) \leq min\text{-}edit(Q') \leq ged(r, s), \forall Q \subseteq Q' \subseteq Q_r \backslash Q_s$.

Proposition 2 (disconnectivity) $min\text{-}edit(Q_1 \cup Q_2) = min\text{-}edit(Q_1) + min\text{-}edit(Q_2), \forall w_i \in Q_1, w_j \in Q_2, w_i$, *and* w_j *have no common vertices.*

4.2 Minimum prefix length

Recall Example 5, although the lower bound of common q-grams is LB_{path}, it is likely the minimum edit operations that occur on mismatching q-grams already exceed τ, and thus the candidate should be discarded. Based on this observation, we seek a *minimum* prefix such that at least $\tau + 1$ edit operations are needed to affect all the prefix q-grams. In this case, r and s are guaranteed to not meet the edit distance constraint if all their prefix q-grams mismatch.

Algorithm 4: MinEditLowerBound (Q)

Input : Q is a set of q-grams.
Output : A lower bound of the minimum edit operations that affect all the q-grams in Q.

```
edit ← compute min-edit(Q) with the greedy algorithm;
return ⌈ edit / (ln|Q| − ln ln|Q| + 0.78) ⌉
```

Algorithm 5: MinEdit (Q)

Input : Q is a set of q-grams.
Output : The exact minimum edit operations that affect all the q-grams in Q.

```
edit ← compute exact min-edit(Q);
return edit
```

Algorithm 6: MinPrefixLen (Q_r)

Input : Q_r is a sorted set of q-grams of graph r.
Output : The minimum prefix length of Q.

```
left ← τ + 1;   right ← τ · D_path(r) + 1;
while left < right do
    mid ← (left + right)/2;
    edit ← MinEditLowerBound(Q_r[1 .. mid]);
    if edit ≤ τ then left ← mid + 1;
    else right ← mid;
right ← left; left ← τ + 1;
while left < right do
    mid ← (left + right)/2;
    edit ← MinEdit(Q_r[1 .. mid]);
    if edit ≤ τ then left ← mid + 1;
    else right ← mid;
return left
```

The monotonicity (Proposition 1) enables us to find the minimum prefix length for a set of q-grams Q_r with a binary search within the range of $[\tau + 1, \tau \cdot D_{path}(r) + 1]$ (Algorithm 6). It performs two rounds of binary search, the first seeking an *upper bound* of the minimum prefix length and the second the exact answer. In the first round, to check whether the q-grams within a prefix length need $\tau + 1$ edit operations, the greedy algorithm (Algorithm 4) is called to find the *lower bound* of the answer to the minimum graph edit operation problem. The result from the first round is used as the upper bound of the second round, in which the exact algorithm (Algorithm 5) is applied iteratively.

Proof (*correctness of Algorithm 6*) The minimum prefix length is at least $\tau + 1$ as an edit operation on a vertex affects at least one q-gram. The minimum prefix length is at most $\tau \cdot D_{path}(r) + 1$, according to Lemmas 1 and 2. Hence, left and right in the first round of binary search bound the minimum prefix length. Since it ends when left equals right, and right is only modified when at least $\tau + 1$ edit operations are needed to affect the q-grams in $Q_r[1 \ldots \text{right}]$, the first round of binary search always returns an upper bound of the minimum prefix length. With the upper bound gained in the first

round, the second round of binary search finds the minimum prefix length, according to Proposition 1. □

Lemma 3 (minimum edit filtering) *For the q-grams of graphs r and s, denote the minimum prefix lengths p_r and p_s, respectively. If $ged(r, s) \leq \tau$, Q_r's p_r-prefix and Q_s's p_s-prefix must share at least one common q-gram.*

To apply minimum edit filtering condition in the basic graph similarity search algorithm, we replace Line 4 in Algorithm 2 with "$p_r \leftarrow \mathsf{MinPrefixLen}(Q_r)$", and Line 4 in Algorithm 3 "$p_s \leftarrow \mathsf{MinPrefixLen}(Q_s)$".

Example 7 Consider graph s in Fig. 1 and its five 1-grams sorted according to the order as they are listed in Example 3. When τ is 1, the minimum prefix length is 2, while the prefix length before using minimum edit filtering is 4.

5 Label filtering

In this section, we introduce another approach to exploit the label differences in mismatching q-grams.

Minimum edit filtering estimates a edit distance lower bound, but works in a pessimistic way assuming edit operations are scattered. However, edit operations can be clustered within several mismatching q-grams.

Example 8 Consider Fig. 1 and $q = 1$. The two mismatching q-grams are bounded in dashed lines. If we compare the labels in the mismatching q-gram in the right bounding box with those in r, they already incur at least one edit operation, because there is no nitrogen atom (N) in r.

Motivated by this idea, we are able to establish a lower bound of edit distance from the labels in mismatching q-grams. Let $L_V(r)$ denote the multiset of the vertex labels in r and $L_E(r)$ the multiset of the edge labels in r.

Lemma 4 (local label filtering) *Consider graphs r and s. If $ged(r, s) \leq \tau$, $\forall r' \sqsubseteq r$, $|L_V(r')\backslash L_V(s)|+|L_E(r')\backslash L_E(s)| \leq \tau$.*

Applying local label filtering on whole graphs immediately yields global label filtering.

Lemma 5 (global label filtering) *Consider graphs r and s. If $ged(r, s) \leq \tau$, $\Gamma(L_V(r), L_V(s)) + \Gamma(L_E(r), L_E(s)) \leq \tau$, where $\Gamma(A, B) = \max(|A|, |B|) - |A \cap B|$.*

According to Lemmas 4 and 5, we prune a graph pair (r, s), if $|L_V(r')\backslash L_V(s)| + |L_E(r')\backslash L_E(s)| > \tau$ for any subgraph r' of r, or $\Gamma(L_V(r), L_V(s)) + \Gamma(L_E(r), L_E(s)) > \tau$.

Although the local label filtering can be applied on any subgraphs, we choose as a heuristic to use it on the subgraphs containing at least one mismatching q-gram, since mismatching q-grams may imply differences in vertex and edge labels. In addition, we employ minimum edit filtering to enhance the power of local label filtering. Recall the disconnectivity of minimum graph edit operations (Proposition 2), the observed mismatching q-grams can be articulated and form a set of connected components. We may derive the lower bound of edit distance on the whole graph by computing that in each component and summing them up.

Algorithm 7: LocalLabelFilter(Q, s)

Input : Q is a set of mismatching q-grams from r to s.
Output : A lower bound of $ged(r, s)$.

```
1 C ← the connected components formed by Q;
2 ε ← 0;
3 for each c_i ∈ C do
4     ε_m ← MinEdit(c_i);
5     ε_l ← |L_V(c_i)\L_V(s)| + |L_E(c_i)\L_E(s)|;
6     ε ← ε + max(ε_m, ε_l);
7 return ε
```

Algorithm 7 explains the implementation of the enhanced local label filtering after including minimum edit filtering. It first computes the connected components of the input q-grams (Line 1). This is implemented with a disjoint set data structure by one scan of the input q-grams. For each connected component consisting of mismatching q-grams, we compute the minimum edit operations within the component using (1) minimum edit filtering (Line 4) and (2) local label filtering (Line 5). The larger one is then chosen as the edit distance lower bound within this component and added up to the total edit distance lower bound. The time complexity of the algorithm is $O(|V_Q|^\tau + q|Q| + |E_Q|)$, where V_Q (resp. E_Q) is the set of vertices (resp. edges) contained by the q-grams in Q. In case of a large $|V_Q|$, we may calculate an approximate answer to the minimum edit operation problem with the greedy algorithm, and the time complexity is $O(\tau(|V_Q| + |Q|) \log |Q| + q|Q| + |E_Q|)$.

Example 9 Consider graphs r and s in Fig. 5, $\tau = 2$, and $q = 2$. Global label filtering yields a lower bound of 2; count filtering needs at least two common q-grams, and they do share C–C–C and C–C–C; minimum edit filtering gives two edit operations. Hence, the pair passes the three filters. The bounded regions indicate two connected components by jointing the mismatching q-grams in r. The number of edit operations on the left is 1 (via minimum edit filtering) and 2 on the right (via local label filtering). Thus, the pair can be pruned.

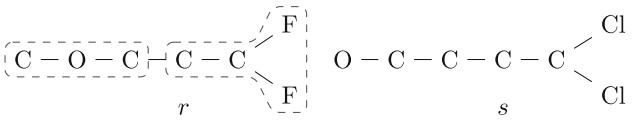

Fig. 5 Example of local label filtering

6 Vertex degree filtering

Recall Condition 1 on matching q-grams. One may notice that by extracting q-grams out of graphs, we neglect certain structural information and compare only the linear structure. This section introduces another filtering technique based on degree information attached to vertices. We provide an edit distance lower bound by comparing two degree sequences and propose a novel q-gram matching condition.

6.1 Exploiting difference in vertex degrees

Consider an instance of two matching q-grams under Condition 1. Since it yields a bijection on their vertices in the original graphs, the degrees must be adjusted to the same using no more than τ edit operations, if they meet the edit distance constraint. Inspired by this, we devise a degree-based filtering scheme to test the degree differences between each pair of vertices in two q-grams sequentially and count how many edit operations are needed to level the degrees. As any two q-grams satisfying Condition 1 have the same label sequence, we are confined to applying the operations that change degrees, namely edge insertions and deletions. The idea of degree filtering is illustrated in the following example.

Example 10 Consider the two q-grams with vertex degrees shown in the parentheses, $\tau = 1$ and $q = 3$.

w_r : $\mathtt{C}_1(3)$–$\mathtt{C}_2(2)$ – $\mathtt{N}_1(3)$ – $\mathtt{C}_3(1)$

w_s : $\mathtt{C}_1(4)$ – $\mathtt{C}_2(3)$ – $\mathtt{N}_1(3)$ – $\mathtt{C}_3(2)$

The two q-grams possess the same label sequence and thus satisfy Condition 1. Comparing the degrees from left to right, it takes at least two edit operations to make the two sets of degrees identical, for example, inserting in w_r an edge between $\mathtt{C}_1$ and $\mathtt{C}_2$, and then inserting an edge to $\mathtt{C}_3$. It is obvious they cannot match under this threshold.

To handle the general case, we first look at *asymmetric* q-grams and the symmetric case will be discussed in the end of this section. Let $deg(w_r)$ denote the degree sequence of w_r, comprising the degrees $deg(w_r[i])$ of the vertices in w_r in sequence, $i \in [1, q+1]$. To level each $deg(w_r[i])$ and $deg(w_s[i])$ of the degree sequences, the following four edge edit operations are available:

- Op. 1: insert a labeled edge $e(u, v)$, $u, v \in V(w_r)$;
- Op. 2: delete a labeled edge $e(u, v)$, $u, v \in V(w_r)$;
- Op. 3: insert a labeled edge $e(u, v)$, $u \in V(w_r)$, $v \in V(r) \backslash V(w_r)$;
- Op. 4: delete a labeled edge $e(u, v)$, $u \in V(w_r)$, $v \in V(r) \backslash V(w_r)$.

Op. 1 and 2 represent the operations on two vertices in this q-gram, while Op. 3 and 4 are the operations involving one vertex in the q-gram and one outside (assuming this vertex can always be found in r). To check whether two q-grams match with respect to the edit distance constraint, we formulate the *minimum edge edit operation* problem.

Problem 5 (*minimum edge edit operation*) Given two q-grams w_r and w_s, find the minimum number of edge edit operations to convert $deg(w_r)$ to $deg(w_s)$.

The above problem can be converted to a minimum cardinality perfect b-matching problem [16] and solved in $O(q^4 \log q)$ time. Next, we consider an additional constraint on this problem and hence compute it more efficiently as well as obtain a filtering strategy with greater pruning power.

6.2 Leveraging existing edges on q-gram vertices

The minimum edge edit operation problem quests for the least edit operations to adjust the degrees of a q-gram, but ignores the existing edges incident on the vertices of the q-gram. We refer as *existing edges* the edges that are not included in the q-gram but whose both ends belong to the q-gram. Let us take the following example.

Example 11 Figure 6 depicts the two q-grams in Example 10, with existing edges between the vertices in r and s, shown in dashed lines. Looking into the edit operations that make r isomorphic to s, where $w_r[i]$ maps to $w_s[i]$ in the resulting bijection, we observe that any solution must contain deleting edge $e(\mathtt{C}_1, \mathtt{N})$ (Op.1) and inserting edge $e(\mathtt{C}_2, \mathtt{C}_3)$ (Op.2). Afterward, the degrees in w_r become 2, 3, 2, and 2, from left to right. To change them to 4, 3, 3, and 2, respectively, three operations in Op. 3 and 4 are required. In all, five edge edit operations are necessary.

The above example showcases that existing edges may incur edit operations. Apart from edge insertion (Op. 1) and deletion (Op. 2), changing edge labels may also happen, since these edges are outside the q-grams, and may differ in labels. To this end, we introduce another edge edit operation.

- Op. 5: change the label of an edge $e(u, v)$, $u, v \in V(w_r)$.

After comparing the two q-grams and the existing edges, it is easy to obtain the necessary operations in Op. 1, 2, and 5. Note these are also the only operations involving a pair of vertices in the q-gram. The degrees in the q-grams change thereafter,

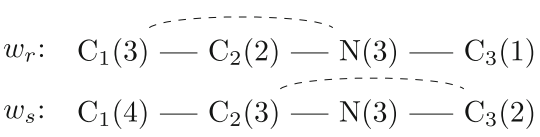

Fig. 6 Idea of exploiting existing edges

Algorithm 8: CheckDegree (w_r, w_s, r, s, τ)

Input : w_r and w_s are two q-grams satisfying Condition 1; r and s are their original graphs; τ is an edit distance threshold.
Output : A boolean indicating whether the w_r and w_s satisfy degree-based matching condition.

```
1  ε ← 0;
2  for each i ∈ [1, q + 1] do
3      Δ_i ← deg(w_r[i]) − deg(w_s[i]);
4  for each i ∈ [1, q] do
5      for each j ∈ [i + 1, q + 1] do
6          if e(w_r[i], w_r[j]) ∈ E(r) and e(w_r[i], w_r[j]) ∈ E(s)
           and l_E(e(w_r[i], w_r[j])) ≠ l_E(e(w_r[i], w_r[j])) then
7              ε ← ε + 1 ;                          /* Op. 5 */
8          if e(w_r[i], w_r[j]) ∉ E(r) and e(w_r[i], w_r[j]) ∈ E(s)
           then
9              ε ← ε + 1 ;                          /* Op. 1 */
10             Δ_i ← Δ_i + 1;   Δ_j ← Δ_j + 1;
11         if e(w_r[i], w_r[j]) ∈ E(r) and e(w_r[i], w_r[j]) ∉ E(s)
           then
12             ε ← ε + 1 ;                          /* Op. 2 */
13             Δ_i ← Δ_i − 1;   Δ_j ← Δ_j − 1;
14 for each i ∈ [1, q + 1] do
15     ε ← ε + |Δ_i| ;                          /* Op. 3 and 4 */
16 if ε ≤ τ then return true
17 else return false
```

and then the new differences in the degrees are eliminated using Op. 3 and 4. The pseudo-code of the algorithm is presented in Algorithm 8, and the degree-based matching condition is summarized in Condition 2.

Condition 2 (degree-based match) *Consider two q-grams w_r and w_s satisfying Condition 1, and threshold τ. w_r matches w_s, if no more than τ edge edit operations are necessary to transform (1) $deg(w_r)$ to $deg(w_s)$, and (1) the existing edges incident on $V(w_r)$ to those incident on $V(w_s)$.*

Algorithm 8 checks whether two q-grams match under Condition 2 in $O(q^2)$ time. It is more efficient than the solution to the minimum edge edit operation problem using b-matching and the pruned candidates subsume all those can be pruned by the latter. To integrate it into the algorithm for graph similarity search queries, we change Line 9 in Algorithm 3 to "$A[r] \leftarrow$ CheckDegree(w_r, w_s, r, s, τ)".

Symmetric q-gram may have *asymmetric* degree sequence. To deal with this case, the pair of q-grams need to be checked from both sides, that is, to run Algorithm 8 on w_r against w_s as well as the inversion of w_s. Either satisfying Condition 2 makes the pair of q-grams match.

In relation to the space cost imposed by utilizing vertex degrees, recall we check Condition 1 with the hash codes of q-grams' label sequences. Supposing a label sequence is hashed into a 4-byte integer, additional $q + 1$ bytes are needed here to record the q-gram's vertex identifiers, which are used to retrieve degrees and existing edges from the data graphs. In all, our algorithm needs a total of $q + 5$ bytes to store a q-gram. In particular, we keep only hash codes in the inverted index; only when the hash codes match, we check Condition 2 using the q-gram sets and the data graphs.

We remark considering the degree information associated with a q-gram implies the effort to qualify the possible matches of a q-gram, which reduces the candidates in return. One may compare q-grams under the matching condition exploiting degree information to positional q-grams [10] in string similarity queries, which only match if their positions in the strings differ by at most τ. Vertex degree filtering is orthogonal to count filtering and prefix filtering, and thus, conducting it after the two existing filters does not affect the correctness. Moreover, it utilizes more structural information to improve the selectivity of q-grams, while retaining the advantage of tighter lower bounds of path-based q-grams over tree-based q-grams regarding count filtering condition.

7 Verification algorithm

The verification comprises (1) multiple filters that prune unpromising candidates and (2) edit distance computation.

7.1 Integrating multiple filters

Algorithm 9 presents the Verify algorithm. The candidates are examined by three filters in succession: global label filtering (Lines 3–4), count filtering (Lines 5–6), and local label filtering (Lines 7–9). We put global label filtering first because it prunes graphs that disagree on labels with a small cost, $O(|V| + |E|)$ for each check, whereas the worst-case complexities of the latter two are $O(|Q|^2 q^2)$ and $O(|V|^\tau + q|Q| + |E|)$, respectively. Count filtering is invoked before local label filtering, since the latter takes as input the set of mismatching q-grams, which is a by-product of count filtering. After the three filters, those still surviving are verified by the expensive edit distance computation.

It is noteworthy to mention the CompareQGrams algorithm in Line 5. Using Conditions 1 and 2, it extracts the *sets* of mismatching q-grams in both r and s, returned in Q'_r and Q'_s, respectively. In addition, it computes the *numbers* of mismatching q-grams in r and s, returned in ε_2 and ϵ_3, respectively. We note the multiple matching of common q-grams is allowed in computing the sets but disallowed in computing the numbers. For instance, if w_r and w'_r both match w_s, one of them, either w_r or w'_r, has to mismatch. As a result, this mismatching q-gram contributes to the number of mismatching q-grams in ε_2 to be tested by count filtering. However, it is not included in Q'_r, because we are unsure whether the mismatching one is w_r or w'_r.

Algorithm 9: Verify(r, A)

```
Input  : s is a query graph; A is map indicating s's candidates.
Output : T = { r | ged(r, s) ≤ τ }.
1  T ← ∅;
2  for each r such that A[r] = true do
3      ε1 ← Γ(L_V(r), L_V(s)) + Γ(L_E(r), L_E(s)) ; /* global
       label filtering */
4      if ε1 ≤ τ then
5          (Q'_r, Q'_s, ε2, ε3) ← CompareQGrams(Q_r, Q_s) ;
           /* count filtering */
6          if ε2 ≤ τ · D_path(r) and ε3 ≤ τ · D_path(s) then
7              ε4 ← LocalLabelFilter(Q'_r, s) ; /* local label
               filtering */
8              ε5 ← LocalLabelFilter(Q'_s, r) ; /* local label
               filtering */
9              if ε4 ≤ τ and ε5 ≤ τ then
10                 edit ← GraphEditDistance(r, s);
11                 if edit ≤ τ then T ← T ∪ { r };
12 return T
```

Algorithm 10: GraphEditDistance(r, s)

```
Input  : r is a data graph; s is a query graph.
Output : ged(r, s), if ged(r, s) ≤ τ; or τ + 1, otherwise.
1  M ← DetermineVertexOrder(r);
2  initial.V_r ← ∅, initial.V_s ← ∅, initial.n = 0;
3  Q.push(initial) ;                  /* a priority queue */
4  while Q ≠ ∅ do
5      current ← Q.pop();
6      if current.n = |V(r)| then
7          return current.g(x)
8      v ← M[current.n + 1];
9      for each v' ∈ V(s) or a dummy vertex, such that
       v' ∉ current.V_s and |deg(v) − deg(v')| ≤ τ do
10         next.V_r ← current.V_r ∪ { v };
11         next.V_s ← current.V_s ∪ { v' };
12         next.n ← current.n + 1;
13         next.g(x) ← ExistingDistance(next);
14         next.h(x) ← EstimateDistance(next);
15         if next.g(x) + next.h(x) ≤ τ then Q.push(next);
16 return τ + 1
```

7.2 Graph edit distance computation

Most widely used exact approaches for computing graph edit distance are based on A* algorithm [11]. We first review a state-of-the-art approach for graph edit distance [19] and then see how our techniques can be employed for speedup.

A* explores the whole possible vertex mapping space between two graphs in a *best-first* fashion. It maintains a priority queue of states such that each state represents a partial vertex mapping, associated with a "priority" via function $f(x)$. $f(x)$ is the sum of two functions: (1) the edit operations observed from the initial state to the current (denoted $g(x)$); and (2) a heuristic estimate of the edit operations that will occur from the current to the goal—a state with all the vertices mapped (denoted $h(x)$). A* guarantees to find the optimal vertex mapping whenever the goal is popped from the queue, provided $h(x)$ is *admissible*, that is, $h(x)$ does not overestimate the distance from the current state to the goal.

With no vertex mapped initially, we form a new state by mapping a vertex of r to either a vertex of s, or none to imply vertex deletion. $g(x)$ is the number of edit operations between the partial graphs regarding the current mapping. For $h(x)$ in weighted graph edit distance, [19] gives an estimation of the edit distance between the remaining parts via bipartite matching. For our unweighted case, $h(x)$ becomes exactly the result of "global" label filtering:

$$h(x) = \Gamma(L_V(r_q), L_V(s_q)) + \Gamma(L_E(r_q), L_E(s_q)),$$

where r_q is constituted of the current unmapped vertices and their incident edges.

Algorithm 10 details the A* algorithm to compute graph edit distance. At first, an order of vertex mapping is determined (Line 1, to be discussed shortly). Starting from an initial state with no vertex mapped, the vertices of r are mapped one by one in the aforementioned order. A new state is formed (Lines 9–12) by including a vertex of r and its counterpart—a dummy vertex to indicate vertex deletion, or an unmapped vertex of s within τ in terms of degree difference. Then, $g(x)$ and $h(x)$ are computed, and the state is inserted to the priority queue thereafter (Line 15). The algorithm terminates when all the vertices of r have been mapped (Line 7), or no vertex mapping within threshold τ is achieved (Line 16).

7.2.1 Optimizing search order

We observe the basic A* algorithm does not discuss the impact of search order on the efficiency of the algorithm. Due to the removal of unpromising candidates with multiple filters, the pairs to be verified are very likely to resemble, though they may not satisfy the edit distance constraint. As for two graphs whose edit distance is not within the threshold, the isomorphic part of the graphs do not incur any edit operations; and therefore, if we start with this part and proceed in a threshold-based manner, the algorithm does not terminate until very late stage of the search process. In contrast, the process ends more quickly if we start with the parts that need edit operations.

Recall the mismatching q-grams identified by CompareQGrams algorithm. The mismatching q-grams indeed contribute edit operations and hence should be favored. Algorithms 11 exploits this idea and determines the order of vertices to be processed by the A* algorithm. The vertices contained by at least one mismatching q-gram are put before the others. In the interest of connectivity, we break tie by mapping vertices in the order of spanning tree, so as to expedite the discovery of edge edit operations. Moreover, we pick as the tree root the vertex with most infrequent label regarding the graph and tie is broken arbitrarily. Using such order

Algorithm 11: DetermineVertexOrder(r, Q'_r)

Input : r is graph; Q'_r is a set of mismatching q-grams from r to s.
Output : An array of vertices that the A* algorithm will find mappings in order.

```
1 M ← [];
2 C ← the connected components formed by Q'_r;
3 for each c_i ∈ C do
4     Insert vertices in c_i into M in the order of spanning tree;
5 Insert the vertices not contained by any mismatching q-gram into
  M in the order of spanning tree;
6 return M
```

leverages the connectivity of a graph and can quickly find edge edit operations. For instance, assume in r, u and v are adjacent vertices in the spanning tree and they are mapped to u' and v' in s, respectively. An edge edit operation occurs if there is no edge between u' and v' in s.

Example 12 Consider graphs in Fig. 7, $q = 1$ and $\tau = 4$. Mismatching q-grams are C$_1$-O, C$_3$-F$_1$, and C$_3$-F$_2$, yielding two connected components as bounded by dashed lines. Thus, we put the vertices in these two components ahead of others and start with C$_1$-O. They are ordered as C$_1$ $\prec$ O, since O is less frequent than C in r. Then, we order the component consisting of C$_3$, F$_1$, and F$_2$. As F is less frequent than C, for example, F$_1$ is picked as root, we have F$_1$ $\prec$ C$_3$ $\prec$ F$_2$ by the order of spanning tree. Finally, we append the remaining vertex C$_2$ and obtain the search order O $\prec$ C$_1$ $\prec$ F$_1$ $\prec$ C$_3$ $\prec$ F$_2$ $\prec$ C$_2$.

7.2.2 Optimizing heuristic estimation

Any lower bound of edit distance can serve as the heuristic estimate $h(x)$ to render the A* algorithm admissible. We consider not only global label filtering but also local label filtering in $h(x)$. The mismatching q-grams in the remaining graphs composed of unmapped vertices are first extracted and then sent into local label filtering to get lower bounds of edit distance between the two remaining graphs. Algorithm 12 provides the pseudocode of the algorithm. Note that we compute mismatching q-grams from both r_q to s_q and s_q to r_q, and hence have two lower bounds from local label filtering. The lower bound from global label filtering is also considered. The maximum of the three is returned as the result of heuristic estimate.

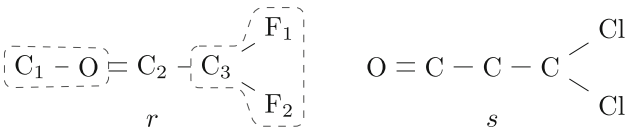

Fig. 7 Example of ordering vertices

Algorithm 12: EstimateDistance(r_q, s_q)

Input : r_q and s_q are two graphs consisting of unmapped vertices.
Output : A lower bound of $ged(r_q, s_q)$.

```
1 ε1 ← Γ(L_V(r_q), L_V(s_q)) + Γ(L_E(r_q), L_E(s_q));
2 (Q'_r, Q'_s) ← CompareQGrams(r_q, s_q);
3 ε2 ← LocalLabelFilter(Q'_r, s_q),
  ε3 ← LocalLabelFilter(Q'_s, r_q);
4 h ← max(ε1, ε2, ε3);
5 return h
```

Algorithm 13: GSimJoin (R, τ)

Input : R is a collection of graphs; τ is an edit distance threshold.
Output : $T = \{\langle r, s\rangle \mid ged(r, s) \le \tau,\ r, s \in R\}$.

```
1  T ← ∅;
2  I_i ← ∅ (1 ≤ i ≤ |U|);            /* inverted index */
3  for each r ∈ R do
4      A ← empty map from id to boolean;
5      Q_r ← r's q-grams sorted in O;
6      p_r ← MinPrefixLen(Q_r);
7      for i = 1 to p_r do
8          w ← Q_r[i];
9          for each s ∈ I_w such that A[s] has not been initialized
           do
10             if abs(|V(r)| − |V(s)|) + abs(|E(r)| − |E(s)|) ≤ τ
               then
11                 A[s] ← true ;   /* find a candidate */
12         I_w ← I_w ∪ {r} ;    /* index for q-gram w */
13     T ← T ∪ Verify(r, A);
14 return T
```

8 Graph similarity join

As a batch version of graph similarity searches, the proposed techniques are ready to be extended to graph similarity join queries. In this section, we introduce the algorithms for self-join first and then R-S join.

8.1 Algorithm for self-join

Combining count filtering, prefix filtering, minimum edit filtering, and local label filtering, we present a graph similarity join algorithm GSimJoin (Algorithm 13). It takes as input a collection of data graphs and follows an index nested loops join style, maintaining an inverted index on the fly. It iterates through each graph $r \in R$. According to Lemmas 2 and 3, the minimum prefix length is calculated by the MinPrefixLen algorithm for each graph r (Line 6). In addition, the numbers of vertices and edges in r and $s \in R$ must differ by at most τ (Line 10). For each q-gram w in Q_r's prefix, it probes the inverted index to find other graphs s that contain w in their prefixes, satisfying Conditions 1 and 2. The candidates are sent into the Verify algorithm. Afterward, r is inserted into w's posting list for future use (Line 12). The

algorithm eventually returns all pairs of graphs $\langle r, s\rangle$ such that $ged(r, s)$ is no more than the given threshold.

8.2 Algorithm for R-S join

The algorithm for joining two different graph databases (Algorithm 14) is designed in index-nested loops join style. It consists of two phases: (1) indexing phase to build an inverted index on the inner relation, and (2) joining phase to scan the outer relation and find join results with the index. The former duplicates the indexing phase of graph similarity search, and the latter is equivalent to invoking the similarity query answering phase for multiple times. As the two relations may differ in size, the efficiency of R-S join is influenced by the choice of inner/outer relations. Assuming both relations and the inverted index fit into main memory, R and S are of the same data distribution, we analyze the join costs with R and S being the inner relation, respectively.

Consider R as the inner relation. The indexing phase computes the prefix length for each $r \in R$ and inserts prefixes into inverted index. The cost of the index phase is

$$C_{index} = |R| \cdot c_p + |R| \cdot l \cdot c_i,$$

where c_p is the cost of generating q-grams of a graph and computing its minimum prefix length, c_i the cost of a posting list insertion, and l the average prefix length.

The cost in the joining phase is divided into three parts: (1) generating q-grams and computing minimum prefixes for graphs in S, (2) probing inverted index to generate candidates, and (3) verifying candidates. The cost of the first part is $|S| \cdot c_p$. The second part is proportional to the total number of inverted index access and thus can be modeled using the frequencies of the q-grams in both R's and S's prefixes. The third part depends on the number of candidates. Let p_R denotes the set of q-grams comprising the prefixes of graphs in R, the cost in the joining phase is

$$C_{join} = |S| \cdot c_p + \sum_{w \in p_R \wedge w \in p_S} c_a + n_c \cdot c_v,$$

where c_a is the cost of an inverted index access, c_v is the cost of one candidate verification, and n_c is the number of candidates to be verified.

Algorithm 14: GSimJoin (R, S, τ)

Input : R and S are two collections of graphs; τ is an edit distance threshold.
Output : $T = \{\langle r, s\rangle \mid ged(r, s) \le \tau, r \in R, s \in S\}$.

```
1 I ← GSimIndex (R, τ) ;                    /* build index */
2 for each s ∈ S do
3     T ← T ∪ GSimQuery(s, I, τ) ;          /* find results */
4 return T
```

Summing up C_{index} and C_{join}, we have the total cost C_R for the case where R is the inner relation. Swapping R and S in the above equations yields the cost for the join with S as the inner relation. The candidate sizes are identical in both cases. Thus, the difference in the two costs is

$$C_R - C_S = (|R| - |S|) \cdot l \cdot c_i.$$

The above equation shows that it is more efficient to choose the smaller one as inner relation to create index for in-memory R-S joins, and the gap becomes more substantial under larger thresholds due to longer prefixes.

Besides index-nested loops join, another possibility to leverage index is to first build inverted indexes on R and S, respectively, and then join the two indexes to find candidates. In this case, the cost of indexing two relations is

$$C'_{index} = |R| \cdot c_p + |R| \cdot l \cdot c_i + |S| \cdot c_p + |S| \cdot l \cdot c_i.$$

In the joining phase, for each identical q-gram w that appears in both p_R and p_S, two inverted lists of w are accessed and merged to derive candidates. Thus, the join cost is

$$C'_{join} = \sum_{w \in p_R \wedge w \in p_S} 2 \cdot c_a + n_c \cdot c_v.$$

Summing up C'_{index} and C'_{join} yields the total cost C_B of the strategy that joining two indexes to derive candidates. Given the identical number of candidates n_c for verification, it is clear that C_B is larger than both C_R and C_S.

9 Subgraph similarity search

This section extends the solution to the problem of subgraph similarity search with edit distance constraints.

For ease of exposition, we define the *subgraph edit distance* from s to r, dented by $sub\_ged(s, r)$, as the minimum number of edit operations that transform s to a graph r' such that r' is subgraph isomorphic to r. The example below illustrates the subgraph edit distance from a graph to another.

Example 13 Figure 8 shows two molecules after omitting hydrogen atoms. Atoms are modeled by vertex labels. Single and double bonds are modeled by edge labels. The subgraph edit distance from s to r $sub\_ged(s, r) = 3$.

Note subgraph edit distance is not symmetric. Given two graphs r and s, $sub\_ged(r, s)$ may not be equal to

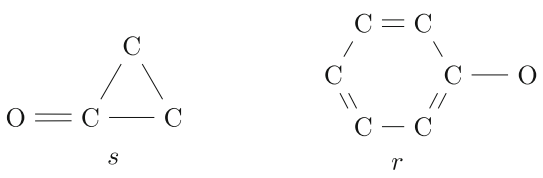

Fig. 8 Example of subgraph edit distance

$sub\_ged(s, r)$. In this paper, the subgraph edit distance constraint is defined from a query graph to a data graph. Next, we first present the algorithmic framework for subgraph similarity search queries, and modify multiple filters afterward, followed by a description of the verification procedure.

9.1 Algorithmic framework

Subgraph similarity search algorithm is composed of two phases. In the indexing phase, it takes as input a collection of graphs and builds an inverted index for data graph q-grams. In the query answering phase, it first decomposes the given query into q-grams and computes its prefix. For each q-gram w in the query's prefix, it probes the index to find candidates that contain w in their q-gram sets. The candidate graphs satisfying size filtering are sent to VerifySub to tell whether they are query results.

For subgraph similarity search, the query may appear (approximately) anywhere in a data graph. In order not to miss any results, we have to track all parts of the data graph. This gives rise to the major difference between subgraph similarity and graph similarity search, that is, the whole q-gram set of each graph is recorded by the inverted index. Nonetheless, prefix filtering still applies to the query graph. The query's prefix length is the same as that for graph similarity queries, and only the q-grams within the query graph's prefix are used to generate candidates. We will see shortly more differences and make necessary modifications.

9.2 Adapting multiple filters

We study how to adapt the filters to subgraph similarity queries, including size filtering, count filtering, minimum edit filtering, local label filtering, and vertex degree filtering.

In the subgraph similarity setting, size filtering removes a data graph if it is smaller than the query by more than τ vertices and edges together, as stated in the following lemma.

Lemma 6 (size filtering for subgraph) *Consider data graph r and query s. If $sub\_ged(s, r) \leq \tau$, $\Lambda(|V(s)|, |V(r)|) + \Lambda(|E(s)|, |E(r)|) \leq \tau$, where $\Lambda(A, B)$ is defined to equal $A - B$, if $A > B$; 0, otherwise.*

On subgraph similarity queries, count filtering only seeks matches for q-grams in Q_s, but not vice versa.

Lemma 7 (count filtering for subgraph) *Consider query graph s and data graph r. If $sub\_ged(s, r) \leq \tau$, s and r share at least $Sub\_LB_{path} = |Q_s| - \tau \cdot D_{path}(s)$ common q-grams.*

Minimum edit filtering looks into the mismatching q-grams Q'_s from Q_s to Q_r, a by-product of the counting process above. Applying MinEdit on Q'_s gives a lower bound.

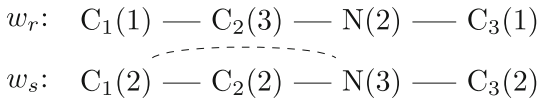

Fig. 9 Example of degree-based match for subgraph

Lemma 8 (minimum edit filtering for subgraph) *Consider query graph s, data graph r, Q'_s as mismatching q-grams from Q_s to Q_r. If $sub\_ged(s, r) \leq \tau$, $min\text{-}edit(Q'_s) \leq \tau$.*

Lemma 9 (label filtering for subgraph) *Consider query graph s and data graph r. If $sub\_ged(s, r) \leq \tau$,*

- *global: $|L_V(s) \backslash L_V(r)| + |L_E(s) \backslash L_E(r)| \leq \tau$; and*
- *local: $|L_V(s') \backslash L_V(r)| + |L_E(s') \backslash L_E(r)| \leq \tau, \forall s' \sqsubseteq s$.*

We apply the global label filtering on whole graphs, and the local label filtering on the connected components of mismatching q-grams. With the mismatching q-grams from Q_s to Q_r, minimum edit filtering and local label filtering are applied to derive two distance lower bounds, respectively. The larger one is chosen as the edit distance lower bound within this component and then summed up.

Vertex degree filtering finds the least edit operations to convert w_s's degree sequence to w'_s such that the degree sequence of w_r is *inclusive* of that of w'_s, that is, $deg(w_r[i]) \geq deg(w'_s[i])$, $i \in [1, q+1]$. We also enforce the existing edges on w_r's vertices to be *inclusive* of those on the vertices of w'_s, that is, $\forall u, v \in w'_s$, if $e(u, v) \in E(s)$, $e(f(u), f(v)) \in E(r)$.

Condition 3 (degree-based match for subgraph) *Consider q-grams w_r and w_s satisfying Condition 1. Given a threshold τ, w_r matches w_s, if no more than τ edge edit operations are needed to make (1) $deg(w_r)$ inclusive of $deg(w_s)$ and (2) existing edges in w_r inclusive of those in w_s.*

Example 14 Consider the q-grams in Fig. 9 and $\tau = 2$. Dashed lines represent the exiting edges of the q-grams. By deleting in w_s the existing edge between C$_1$ and N, and deleting one edge at C$_3$, we can change $deg(w_s)$ to 1, 2, 2, 1, included by $deg(w_r)$: 1, 3, 2, 1. It takes only two edge edit operations, and thus w_r matches w_s consequently.

To test whether w_r and w_s match under Condition 3, we use Algorithm 8 and make the following modifications: (1) We first consider the existing edges in w_s and apply necessary Op. 1, 2, and 5, resulting a new degree sequence w'_s. (2) We sum up the differences where $deg(w'_s[i])$ is greater than $deg(w_r[i])$, meaning that only Op. 4 is applied. The results are summed up and then compared with τ.

We summarize the verification for subgraph similarity search in Algorithm 15. The algorithm accepts as input a query graph and iterates to verify one candidate at a time.

Algorithm 15: VerifySub(s, A)

```
   Input  : s is a query graph; A is map indicating s's candidates.
   Output : T = {r | sub_ged(s, r) ≤ τ}.
1  T ← ∅;
2  for each r such that A[r] = true do
3      ε1 ← |LV(s)\LV(r)| + |LE(s)\LE(r)|;
4      if ε1 ≤ τ then
5          (Q's, ε2) ← CompareQGrams(Qs, Qr);
6          if ε2 ≤ τ · Dpath(s) then
7              ε3 ← LocalLabelFilter(Q's, r);
8              if ε3 ≤ τ then
9                  edit ← SubgraphEditDistance(s, r);
10                 if edit ≤ τ then T ← T ∪ {r};
11 return T
```

Global label filtering (Lines 3–4), counting filtering (Lines 5–6), and local label filtering (Lines 7–8) are employed successively. The CompareQGrams algorithm in Line 5 extracts the mismatching q-grams from Q_s to Q_r, returned in Q'_s, as well as its number in ε_2. Surviving candidates are verified through the final subgraph edit distance computation.

We use the A*-based algorithm to handle the final verification with a few minor modifications. The mismatching q-grams from s to r are collected to determine the order of vertex mapping, and the adapted label filtering for subgraph is utilized to deliver an improved estimation of $h(x)$.

10 Extensions

This section discusses the extension to directed multigraphs.

A *directed multigraph*, or *multidigraph*, $r = (V, E, l_V, l_E)$ is a labeled graph such that (1) the edges are directed; (2) multiple edges may exist between vertices; and (3) self-loops may exist on vertices. Apart from the edit operations defined in Sect. 2.1, for multidigraphs, we allow another operation: Change the direction of an edge.

The aforementioned techniques can be directly applied except for the q-gram extraction of a multidigraph. The basic idea is to convert it to a simple graph, generate q-grams, and then recover multiple directed edges and self-loops.

Example 15 Figure 10 sketches a multidigraph r. Edge labels are omitted, and subscripts are added to the carbon atoms for ease of exposition. First, we construct an *underlying simple graph* $\bar{r}$ of r by

- replacing multiple directed labeled edges between vertices with one single undirected unlabeled edge, and
- discarding self-loops at vertices.

For instance, the three directed edges between C_1 and C_3 in r are converted to one undirected edge in $\bar{r}$, the directed edges between C_2 and C_3 are changed to one undirected edge, and

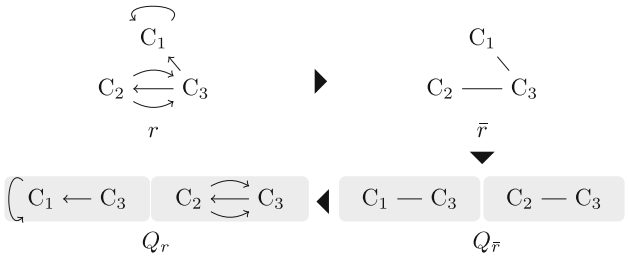

Fig. 10 Example of extracting multidi q-grams

the self-loop at C_1 is discarded. Then, we extract path-based q-grams from $\bar{r}$. Two q-grams are obtained as shown in $Q_{\bar{r}}$. Finally, for each q-gram, we recover the directions, multiple edges, and self-loops that exist in the multidigraph,for example, in C_2-C_3, multiple directed edges replace the single edge; C_1-C_3 is oriented, and a self-loop is assigned to C_1, as shown in Q_r.

To distinguish the two types of q-grams, the q-grams extracted from the underlying simple graph are called "simple q-grams", and the q-grams with directions, multiple edges, and self-loops recovered are called "multidi q-grams". Next, we present the matching condition of multidi q-grams. As we employ simple q-grams as intermediates to obtain multidi q-grams, the vertex sequence of a multidi q-gram is the same as that of its simple q-gram. Hence, to compose the multidi q-gram's label, sequence is straightforward. Denote $deg^-(w_r)$ (resp. $deg^+(w_r)$) the in-degree (resp. out-degree) sequence of w_r, comprising the in-degrees $deg^-(w_r[i])$ (resp. $deg^+(w_r[i])$) of the vertices in w_r, $i \in [1, q+1]$.

Condition 4 (multidi q-gram matching condition) *Given a threshold τ, two multidi q-grams w_r and w_s match, if*

- *w_r and w_s are isomorphic; and*
- *if no more than τ edit operations are needed to convert (1) $deg^-(w_r)$ to $deg^-(w_s)$, (2) $deg^+(w_r)$ to $deg^+(w_s)$, and (3) the existing edges incident on w_r's vertices to those incident on w_s's vertices.*

Thanks to the label sequences of the multidi q-grams, the isomorphism test can be done in $O(|E_r| + |E_s|)$ time, where E_r and E_s are the edges in r and s, respectively. We first label compare the sequences of the two q-grams (hash codes can be used here for $O(1)$ check). If the label sequences are identical, we check whether the directions, multiple edges, and self-loops contained in both q-grams are the same through a sequential scan on both q-grams.[3] After the isomorphism test, we check whether they are degree-based matching using the technique presented in Sect. 6. Note that in-degrees and

[3] Two sequential scans on w_s if the sequences are symmetric.

out-degrees are treated separately, and the returned operation numbers are summed up to be compared with τ.

Count filtering is based on the minimum number of common q-grams after applying τ edit operations. We observe that the edit effects of operations on multidi q-grams are equivalent to those on simple q-grams. Let Q_r denote the set of r's multidi q-grams, Q_r^v denote the multidi q-grams containing vertex u, and Q_r^{uv} the multidi q-grams containing consecutive vertices u and v. In particular,

- inserting a vertex or an edge affects no multidi q-grams;
- deleting a vertex v, changing its label, inserting or deleting a self-loop incident on the vertex, or changing the label or direction of the vertex's self-loop, affect $|Q_r^v|$ multidi q-grams; and
- deleting an edge $e(u, v)$, changing its label or direction, affect $|Q_r^{uv}|$ multidi q-grams.

As $|Q_r^{uv}| \leq |Q_r^u| \leq \max_{u \in V(r)} |Q_r^u|$, the maximum number of multidi q-grams that can be affected by one edit operation is $D_{multidi}(r) = \max_{u \in V(r)} |Q_r^u|$. The lower bound of common multidi q-grams for count filtering is

$$LB_{multidi} = \max(|Q_r| - \tau \cdot D_{multidi}(r), |Q_s| - \tau \cdot D_{multidi}(s)).$$

Since a multidi q-gram is constructed from its corresponding simple q-gram, they contain exactly the same sequence of vertices. Thus, the lower bound $LB_{multidi}$ equals the lower bound LB_{path} derived on its underlying simple graph.

To apply minimum edit filtering, we need to solve the minimum edit operation problem on mismatching multidi q-grams. We may apply vertex label substitutions on the mismatching multidi q-grams and obtain the minimum number of edit operations required to affect all of them. We also apply the local label filtering on the mismatching multidi q-grams, and the filtering rationale remains the same as simple graphs. Similar to count filtering, we note that the results of these filtering techniques are equivalent to those obtained by invoking them on the underlying simple graph.

All the candidates passing the multiple filters are verified by Algorithm 10, except that differences in directions, multiple edges, as well as self-loops are added up to $g(x)$ and estimated in $h(x)$ when a vertex is mapped.

11 Experiments

In this section, we report experimental results and analyses.

11.1 Experiment setup

Two publicly available real-life datasets were used:

Table 1 Dataset statistics

Dataset	$\|R\|$	Avg $\|V\|$	$\|l_V\|/\|l_E\|$	d	Min/max (°)
AIDS	4,000	25.76	44/3	0.08	0/12
PROTEIN	600	32.63	3/5	0.12	0/9

- **AIDS** is an antivirus screen compound dataset from the Developmental Therapeutics Program in NCI/NIH.[4] It contains 42,687 chemical compounds. We randomly sampled 4,000 graphs to make up the set of data graphs.
- **PROTEIN** is a protein database from the Protein Data Bank,[5] constituted of 600 protein structures. Vertices represent secondary structure elements and are labeled with their types—helix, sheet, and loop. Edges are labeled with lengths in amino acids.

Statistics of the datasets are listed in Table 1. The graph density d, defined as $\frac{2|E|}{|V|(|V|-1)}$, influences the number of path-based q-grams. The greater the graph density, the more path-based q-gram in a graph. The maximum degree implies the edit effect of a single edit operation.

Besides real-life datasets, synthetic datasets were generated. The synthetic graph generator[6] measures graph size by the number of edges. The graph density is 0.3 by default, and the cardinalities of vertex and edge label domains are set to 2 and 1, respectively. We applied these *default settings* if not otherwise specified.

We randomly sampled 100 graphs from data graphs and added a random number of edit operations within $[0, \tau]$ to make up the corresponding sets of query graphs.

All the experiments were carried out on a machine of Quad-Core AMD Opteron Processor 8378@800 MHz with 96 GB RAM, running Ubuntu 10.04.1 LTS. All the algorithms were implemented in C++ and ran in main memory.

We measured (1) average prefix length; (2) index size; (3) index construction time, including q-gram extraction, prefix length computation, and inverted list construction; (4) candidates identified by inverted index and surviving size filtering (denoted **Cand-1**); (5) candidates that need *ged* computation (denoted **Cand-2**); and (6) query response time, including candidate generation time (query's q-gram extraction included) and *ged* computation time. Cand-1, Cand-2, and query response time were logged and reported on the basis of 100 search queries unless otherwise specified.

[4] http://dtp.nci.nih.gov/docs/aids/aids_data.html.

[5] http://www.iam.unibe.ch/fki/databases/iam-graph-database/download-the-iam-graph-database.

[6] http://www.cse.ust.hk/graphgen/.

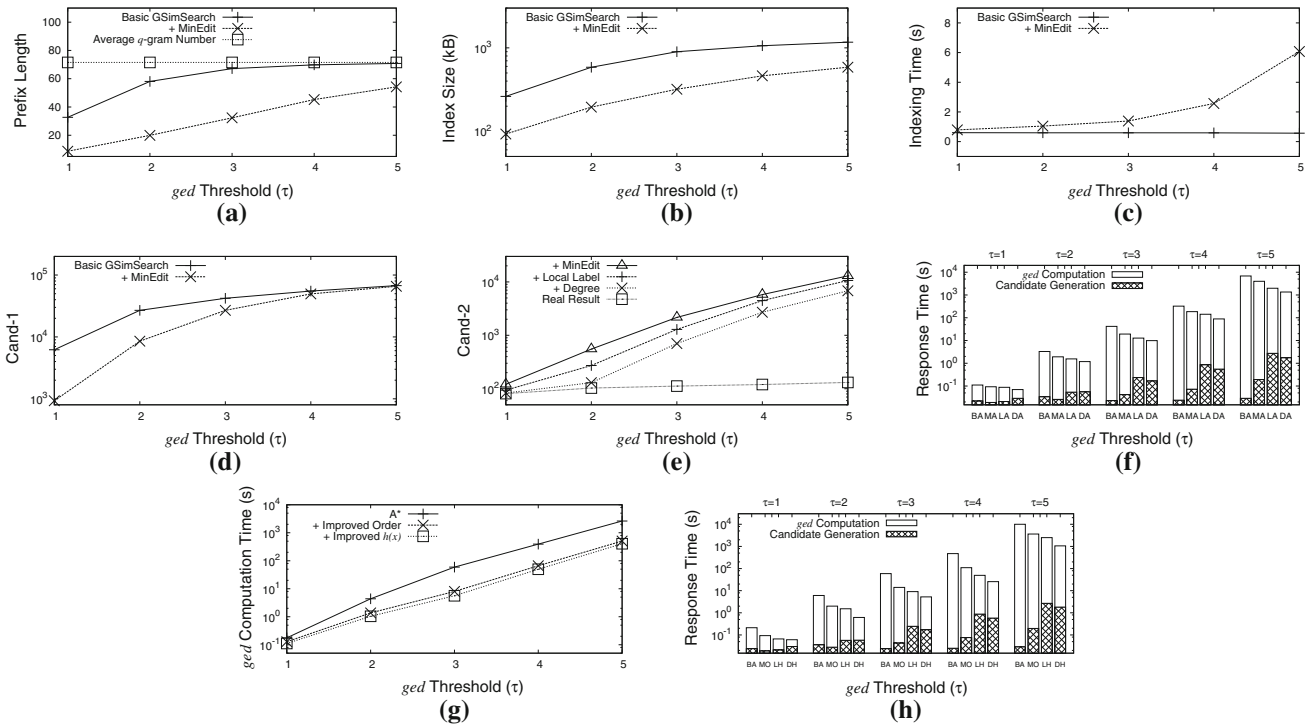

Fig. 11 Effect of filters and *ged* computation. **a** AIDS, prefix length. **b** AIDS, index size. **c** AIDS, indexing time. **d** AIDS, Cand-1. **e** AIDS, Cand-2. **f** AIDS, query response time. **g** AIDS, distance computation time. **h** AIDS, query response time

11.2 Evaluating filtering methods

In order to evaluate the effectiveness of our filtering techniques, we use "Basic GSimSearch" for the GSimSearch algorithm without minimum edit, local label, or vertex degree filtering. "+ MinEdit" denotes applying minimum edit filtering to compute prefix length. "+ Local Label" denotes further applying local label filtering. "+ Degree" denotes further applying degree-based q-gram match condition, that is, the complete filtering of GSimSearch algorithm.

We first study the effect of minimum edit filtering. Figure 11(a) shows the average prefix length of Basic GSimSearch and + MinEdit on AIDS dataset with $q = 4$ and varying edit distance threshold. + Local Label and + Degree have the same prefix length as + MinEdit and thus are omitted in this figure. The prefix lengths of both algorithms grow steadily when the threshold increases. Basic GSimSearch's average prefix length approaches the average number of q-grams in a graph when $\tau > 4$, as most graphs become underflowing. After applying minimum edit filtering, the prefix length is substantially shortened, up to 75 %.

As index size is influenced by prefix length, we plot the memory consumption for index storage in Fig. 11b. Both algorithms need small amount of memory and exhibit a similar trend as on prefix length. The memory consumed by + MinEdit is only 586.9 kB when τ is as large as 5. Figure 11c gives indexing time. We observe a nearly constant indexing time for Basic GSimSearch, and a growing trend for + MinEdit, taking 6.0s at $\tau = 5$. This is expectable, since + MinEdit solves the NP-hard minimum graph edit operation problem for minimum prefix length. We will see shortly this cost is rewarding in the query processing phase.

The number of Cand-1 is mainly influenced by prefix length, as plotted in Fig. 11d. The Cand-1 size can be reduced by as much as 85 % when $\tau = 1$. As for local label and vertex degree filtering, Fig. 11e compares the number of Cand-2 produced by + MinEdit, + Local Label, and + Degree on AIDS. The number of real results is also shown in the figure. Local label filtering results in remarkable reduction on Cand-2, up to 51 %. Utilizing degree information leads to an additional 51 % reduction.

To reflect the effect of filters on running time, we appended the A* algorithm [19], labeled as "A*", to verify the candidates. The overall query response time is plotted in Fig. 11f, where embraces the following combinations:

- BA: Basic GSimSearch / A*;
- MA: + MinEdit / A*;
- LA: + Local Label / A*;
- DA: + Degree / A*.

The overall runtime decreases when we apply more filters, as fewer candidates are sent to verification, with slight increase in candidate generation time though. The maximum speedup

of DA is 2.3x over BA, 2.1x over MA, and 1.8x over LA. We also observe DA has even smaller candidate generation time than LA when $\tau > 2$. This is because degree-based matching condition reduces the candidates to be checked by local label filter, which is shown to be relatively more costly.

11.3 Evaluating graph edit distance computation

To evaluate *ged* computation, we verify with three algorithms the candidates of + Degree under $q = 4, \tau = 4$. Based on "A*", we improve the search order leveraging mismatching q-grams, consequent algorithm labeled "+ Improved Order". Local label filtering is further applied for estimating $h(x)$, consequent algorithm labeled "+ Improved $h(x)$".

Figure 11g reports the overall *ged* computation time to verify the same set of candidate pairs. The optimizations improve the time efficiency of *ged* computation, and the margin gets more significant under larger τ.

Combining the filtering and *ged* computation algorithms according to various techniques employed, we show the overall query response time decomposed into two phases in Fig. 11h. The notations denote the following combinations:

- BA: Basic GSimSearch / A*;
- MO: + MinEdit / + Improved Order;
- LH: + Local Label / + Improved $h(x)$;
- DH: + Degree / + Improved $h(x)$.

BA and MO have small candidate generation time, but become uncompetitive for large τ on total response time, due to large Cand-2 size and inefficient *ged* computation. LH can be up to 2.2x faster than MO and 23.7x faster than BA. DH further reduces LH's running time by up to 59 %. As a controlled experiment, we argue that the performance boost from MO to LH comes from the more effective filtering and efficient verification, and the tighter q-gram matching condition results in the enhancement from LH to DH.

11.4 Evaluating q-gram length

Figures 12a–e present the results of GSimSearch on AIDS with varying q-gram length. As expected, the larger q is, the more q-grams are affected by one edit operation, and hence the longer prefix is. Accordingly, more time is needed to construct the index. Regarding Cand-1 and Cand-2, the general trend is the candidate sizes first drop with increasing q-gram length, reach the bottom at q of 3 and 4, and then rebound. There are several factors contributing to this: (1) Small q indicates a small q-gram domain, and hence the posting list of a q-gram can be fairly long. This leads to a large candidate size, for example, when $q = 2$. (2) Large q indicates a long prefix length. We have to probe more posting lists, which also increases the candidate size. The second factor explains why the candidate sizes rebound for larger q.

The trend of candidate size reflects the query response time for varying q. The figure shows $q = 4$ achieves the best runtime performance for $\tau \in [2, 5]$. The only exception is, when $\tau = 1$, $q = 2$ is the most efficient. This is because the candidate generation of 4-grams is more costly than that of 2-grams, while candidate sizes are very close at $\tau = 1$.

After performing similar tests on PROTEIN, we chose, as *default parameter settings* in the remaining experiments, $q = 4$ on AIDS and $q = 3$ on PROTEIN for GSimSearch.

11.5 Graph similarity searches

This subsection evaluates graph similarity search queries by comparing our algorithms with three alternatives.

11.5.1 Comparison with κ-AT-Search and SEGOS

We compare GSimSearch with κ-AT-Search and SEGOS on both real datasets.

- **GSimSearch** is our proposed algorithm that utilizes path-based q-grams for graph similarity search queries.
- **κ-AT-Search** is a state-of-the-art algorithm based on tree-based q-grams, known as κ-AT's [27]. We reengineered this algorithm and further applied prefix filtering, size filtering, and global label filtering to find the candidates, and basic A* algorithm was used to verify the candidates. we choose q as 1 because it yields the best runtime performance in this set of experiments.
- **SEGOS** is another state-of-the-art algorithm based on star structure [31]. We received the source code from the authors and implemented the basic A* algorithm to verify candidates. Edge labels were not supported and thus discarded where SEGOS was involved. SEGOS is parameterized by (1) k, which defines the top-k star search in TA stage and (2) h, which instructs to perform prune test for every h accessed entries in CA stage. We tuned and chose $k = 100$ and $h = 1{,}000$ for best performance.

First, we compare GSimSearch with κ-AT-Search. In Fig. 13a, b are the average prefix lengths of GSimSearch and κ-AT-Search on AIDS and PROTEIN, respectively. In spite of longer prefix on AIDS, GSimSearch has more average number of q-grams in a graph. For example, κ-AT-Search's prefix length is 8.2 when τ is 1 and the average number of q-grams in a graph is 25.6, while GSimSearch's prefix length is 8.9 and the average number of q-grams is 71.5. This means κ-AT-Search requires two graphs to have an average of $25.6 - 8.2 + 1 = 18.4$ common q-grams to become a candidate, while GSimSearch needs an average of $71.5 - 8.9 + 1 = 63.6$ common q-grams. In this sense,

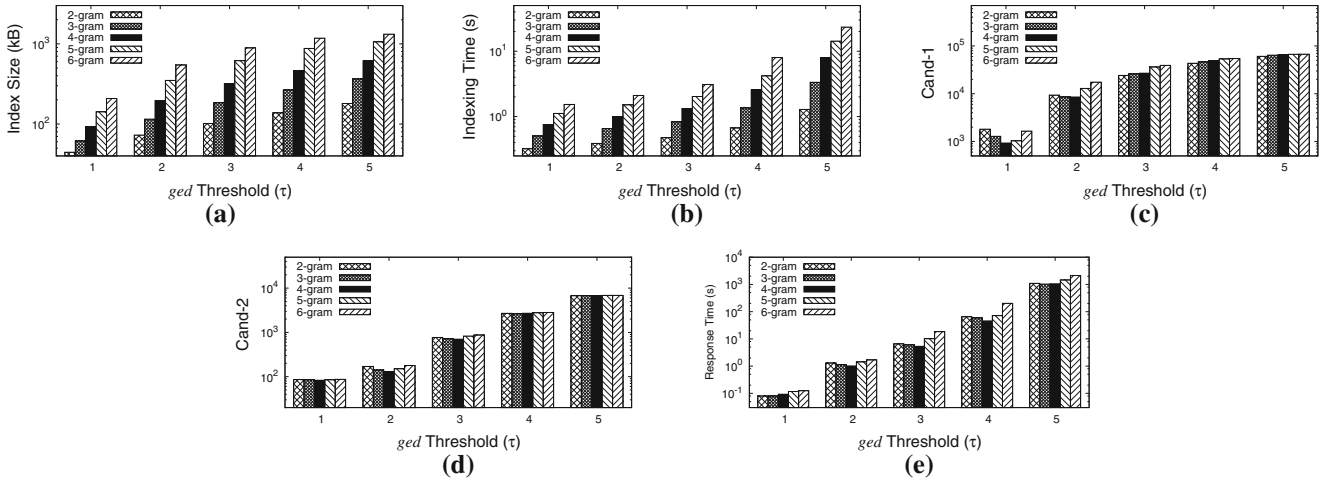

Fig. 12 Effect of q-gram length. **a** AIDS, prefix length. **b** AIDS, indexing time. **c** AIDS, Cand-1. **d** AIDS, Cand-2. **e** AIDS, query response time

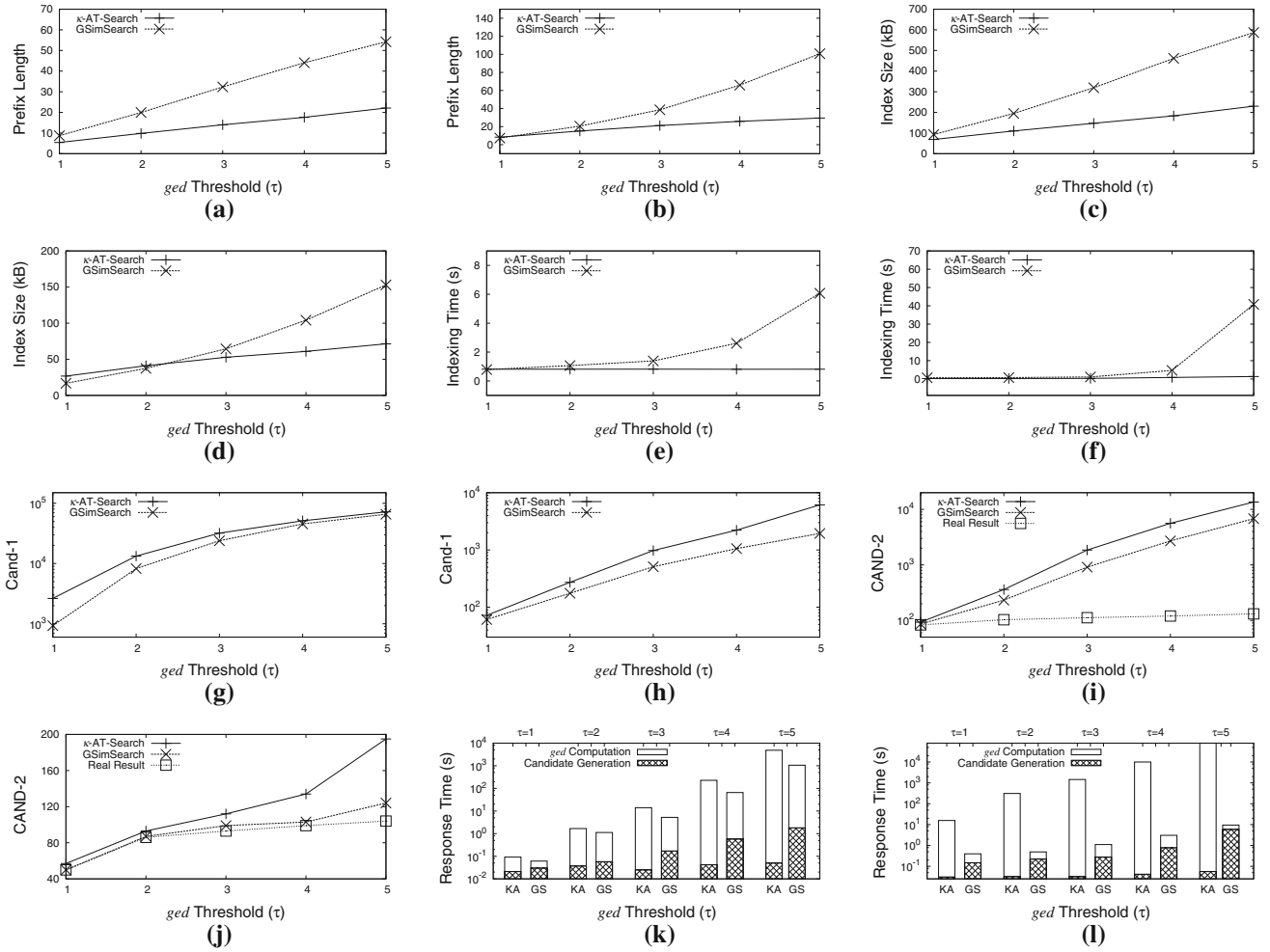

Fig. 13 Comparison with κ-AT-Search. **a** AIDS, prefix length. **b** PROTEIN, prefix length. **c** AIDS, index size. **d** PROTEIN, index size. **e** AIDS, indexing time. **f** PROTEIN, indexing time. **g** AIDS, Cand-1. **h** PROTEIN, Cand-1. **i** AIDS, Cand-2. **j** PROTEIN, Cand-2. **k** AIDS, query response time. **l** PROTEIN, query response time

GSimSearch has a tighter count filtering lower bound. Note the q-gram length is 1 for κ-AT-Search, that is, the count filtering lower bound is the tightest among all its q settings.

Both algorithms are competitive in index size, shown in Fig. 13c, d. GSimSearch consumes more construction time, as in Fig. 13e, f. We also note although the worst-case complexity of extracting paths is $O(|V|\gamma^q)$ for GSimSearch, the time for extracting q-grams is 0.25 s on AIDS and 0.19 s on PROTEIN. Compared with total indexing time, for example, 2.6 s on AIDS and 4.8 s on PROTEIN when $\tau = 4$, the overhead of extracting q-grams is small.

Figure 13g–j gives the Cand-1 and Cand-2 sizes of the two algorithms. GSimSearch performs better than κ-AT-Search on both Cand-1 and Cand-2 sizes. There are three major factors: (1) 4-grams based on paths are more selective than 1-grams based on trees. This results in less number of Cand-1 for GSimSearch. (2) GSimSearch's count filtering constraint is stricter than κ-AT-Search's. (3) GSimSearch employs local label filtering and degree-based matching condition to further prune candidates. The last two factors contribute to GSimSearch's advantage on Cand-2. The running time of both algorithms are shown in Fig. 13k, l ("KA" and "GS" are short for κ-AT-Search and GSimSearch, respectively). The query response time grows rapidly when more edit operations are allowed. κ-AT-Search exhibits better candidate generation time, as GSimSearch applies extra filters to prune candidates. However, GSimSearch is always better than κ-AT-Search in terms of overall query response time, and the gap is more substantial under large τ. The speedup against κ-AT-Search on AIDS is up to 3.5x and 6672.4x on PROTEIN. The latter showcases the superior time advantage of GSimSearch on denser graphs. Section 11.8.2 provides more comparison on graph density.

Next, we compare GSimSearch with SEGOS. The comparisons with SEGOS on index size, indexing time, Cand-2, and query response time are provided in Fig. 14a–h ("SE" and "GS" are short for SEGOS and GSimSearch, respectively). As shown in Fig. 14a, b, both algorithms build space-efficient indexes. GSimSearch consumes less memory but more time to build index.

The two algorithms show a similar increasing trend on Cand-2. On AIDS when $\tau = 1$, SEGOS has a smaller number of Cand-2 than real results, because it derives an edit distance upper bound to confirm certain results without verification. Nevertheless, GSimSearch has a smaller growth rate than SEGOS when τ gets larger. GSimSearch is always faster than SEGOS, with speedup up to 11.9x on AIDS and 1243.8x on PROTEIN. The overall performance superiority boils down to two facts: (1) The filtering techniques in GSimSearch return fewer candidates for most parameter settings. (2) The improved verification reduces running time, and such effect is more remarkable on denser graphs.

In summary, we analyze runtime speedup versus space costup. The results for $\tau = 3$ are listed in Table 2. We compare the inverted index and q-gram set sizes of all algorithms ("q-gram set size" of SEGOS is the size of star representations of data graphs). GSimSearch spends the most amount of space, 1.5x against SEGOS and slightly greater than κ-AT-Search on AIDS, respectively. On the other hand, GSimSearch is 2.7x faster than κ-AT-Search and 2.4x faster than SEGOS under the setting. On PROTEIN, the space consumption of GSimSearch is 3.5x that of κ-AT-Search and 5.0x that of SEGOS, whereas its speedups are 1316.5x and 109.3x, respectively. We suggest investing moderately more memory for runtime speedup if space is not critical.

11.5.2 Comparison with M-tree

We compare GSimSearch with M-tree, a general indexing technique for metric space similarity search [6], on synthetic datasets. Implementation details of M-tree are supplied in the supplementary material. We provide the results on synthetic datasets of $|R| = 100$ in Fig .15. The query response time is measured on the basis of 10 queries, and the queries were sampled from data graphs with random numbers of edit operations added therein.

First, we compare them on the dataset of graph size 10 varying τ, and $q = 2$ was chosen for GSimSearch. We compare the indexing performance in Fig. 15a, b. M-tree takes much longer time, up to four orders of magnitude greater than GSimSearch, to build index due to its ged evaluation between pairs of data graphs. Both algorithms build small indexes, less than 8kB. GSimSearch has a even smaller index size then M-tree when $\tau \leq 2$.

The online performance comparison is shown in Fig. 15c, d. GSimSearch always has fewer Cand-2 than M-tree. As a consequence, the query response time of GSimSearch is constantly smaller than M-tree, with the largest gap being four orders of magnitude. The large gap on running time is attributed to the loosely bounded distance evaluations invoked by M-tree, while GSimSearch runs all verifications in a threshold-based manner.

We also test the indexing scalability on four datasets with graph size ranging in $\{5, 10, 15, 20\}$, and τ fixed to 2. We chose for GSimSearch q equal to 1, 2, 3, 4 for the four sizes, respectively, and show the results in Fig. 15e, f. The indexing time of M-tree, having a much larger starting point, grows faster than GSimSearch. This tendency implies, due to the huge cost of edit distance evaluations, M-tree becomes impractical when graphs are large. The index sizes of both algorithms showcase a growing trend. GSimSearch has smaller indexes for small graphs but with a faster growth rate, and hence larger index sizes for large graphs.

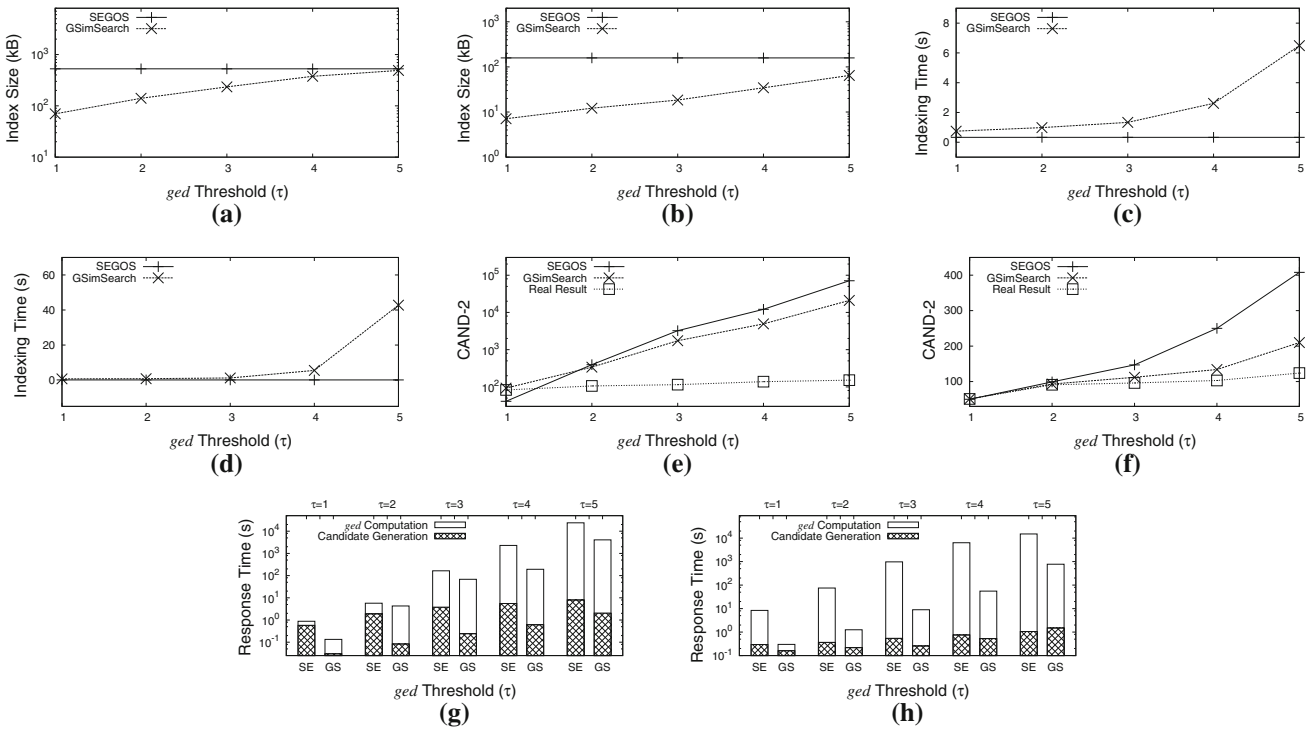

Fig. 14 Comparison with SEGOS. **a** AIDS, index size. **b** PROTEIN, index size. **c** AIDS, indexing time. **d** PROTEIN, indexing time. **e** AIDS, Cand-2. **f** PROTEIN, Cand-2. **g** AIDS, query response time. **h** PROTEIN, query response time

Table 2 Data structure sizes (kB)

	Index	q-Gram multiset	Total
(a) AIDS, $\tau = 3$			
GSimSearch ($q = 4$)	202.3	2,514.0	2,716.4
κ-AT-Search ($q = 1$)	130.3	2,518.3	2,648.8
SEGOS	515.1	1,259.1	1,774.2
(b) PROTEIN, $\tau = 3$]			
GSimSearch ($q = 3$)	51.1	2,604.8	2,655.9
κ-AT-Search ($q = 1$)	33.8	735.5	769.3
SEGOS	158.3	367.8	526.1

11.6 Graph similarity joins

We compare the following algorithms on real datasets without edge labels for similarity joins.

- **GSimJoin** is our proposed algorithm that utilizes path-based q-grams for graph similarity join queries.
- **κ-AT-Join** is an adapted algorithm from κ-AT-Search using tree-based q-grams [27], with $q = 1$.
- **SEGOS-Join** is adapted from SEGOS [31] using star structure. In order to make SEGOS support self-joins, we ran SEGOS in an index-nested loops join mode. It iterates through the dataset and selects each graph as a query with the corresponding database contains all the graphs with smaller identifiers than that of the query.

11.6.1 Self-joins

As a batch version of similarity search queries, similarity join exhibits the same indexing behavior as similarity search. Therefore, we omit the indexing performance comparisons, but include the indexing time in the total running time.

Figure 16a–d reports the number of Cand-2 and running time of the three algorithms on the two datasets. Similar trends are observed as in the experiment for graph similarity searches. GSimJoin outperforms SEGOS-Join in terms of Cand-2 under all the threshold settings except for $\tau = 1$ on PROTEIN. The exception is due to the upper bound validation in SEGOS-Join. Both algorithms generate much fewer candidates than κ-AT-Join. κ-AT-Join is also the slowest under most of the threshold settings, as expected from its largest candidate size. GSimJoin always exhibits less total running time than the others, despite greater indexing time. In particular, GSimJoin is faster than the runner-up SEGOS-Join by up to 58.7x on AIDS and 28.3x on PROTEIN.

11.6.2 R-S joins

We made two relations from the AIDS corpus: the sample of size 4 k used in previous experiments was taken as relation R, and we randomly sampled graphs from the remaining corpus to constitute S. We first fixed the size of S as 20 k and show the running time of GSimJoin under varying τ in Fig. 16e ("IR" and "IS" represent using R and S as inner relation,

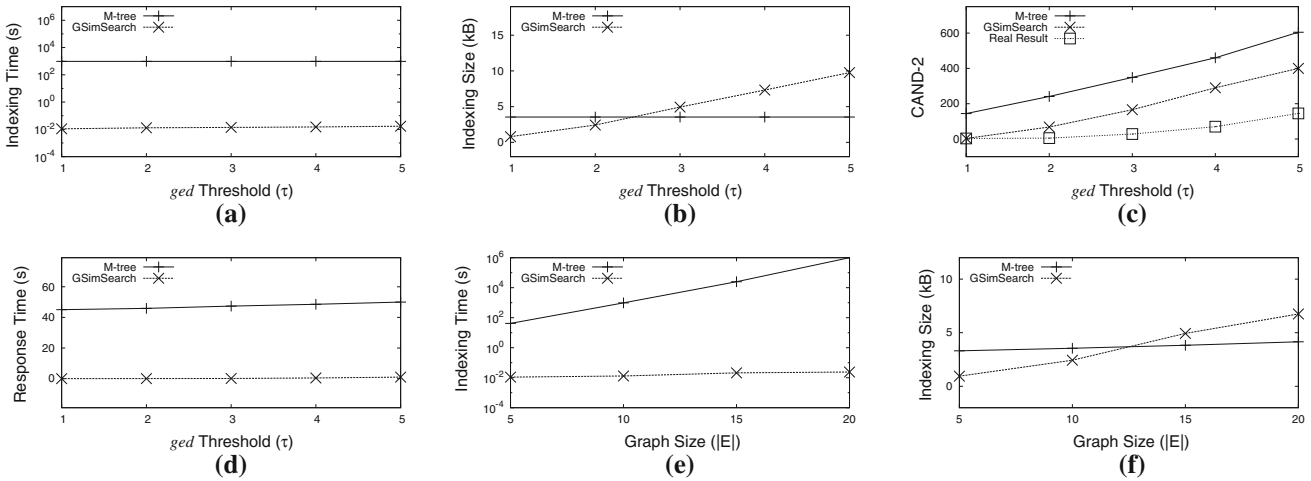

Fig. 15 Comparison with M-tree. **a** Synthetic, indexing time. **b** Synthetic, index size. **c** Synthetic, Cand-2. **d** Synthetic, query response time. **e** Synthetic, indexing time. **f** Synthetic, index size

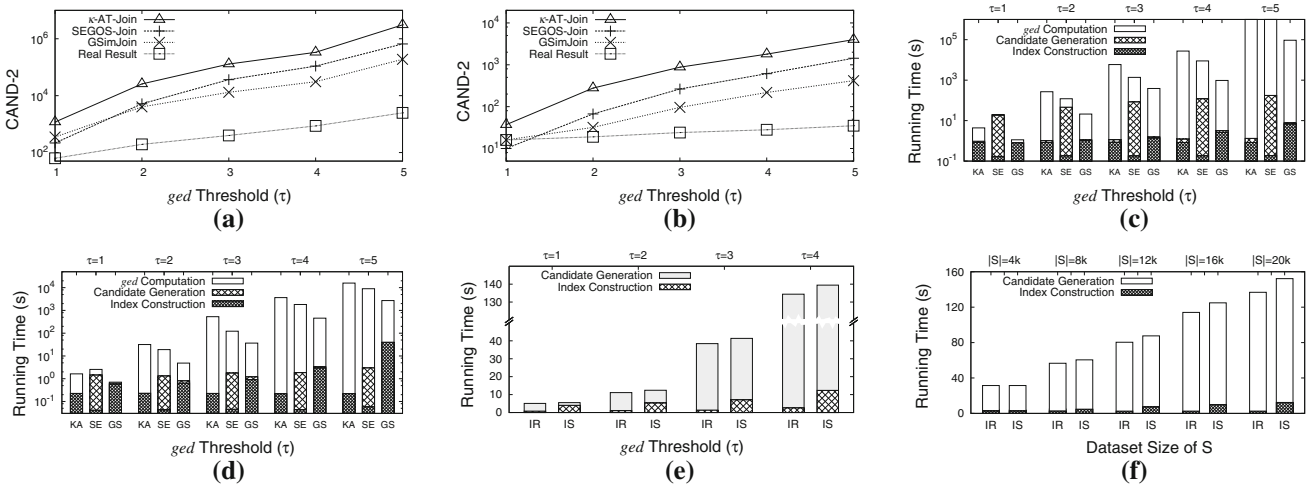

Fig. 16 Graph similarity join. **a** AIDS, Cand-2. **b** PROTEIN, Cand-2. **c** AIDS, total running time. **d** PROTEIN, total running time. **e** AIDS, filtering time. **f** PROTEIN, filtering time

respectively). Since both generate the same candidates and spend the same amount of time on verification, we remove verification time from the figure to make the differences visible. It can be seen that IR saves overall running time by up to 10.5 %, and the gap is increasing with τ. The result corroborates the claim in Sect. 8.2 that indexing the smaller relation yields better runtime performance.

We then fixed τ as 4 and plot in Fig .16f the running time with $|S|$ ranging in {4k, 8k, 12k, 16k, 20k}. IR always beats IS by a small margin, and the gap grows steadily with larger $|S|$, as expected from the analysis in Sect. 8.2.

– **SGSimSearch** is our proposed algorithm that utilizes path-based q-grams for subgraph similarity search queries. $q = 2$ on both AIDS and PROTEIN.
– **AppSub** is an algorithm computing lower bound of *constrained* subgraph edit distance based on star structure [38]. Note the edit operation of changing vertex label is *disabled* in AppSub. The results are consequently a *subset* of that of SGSimSearch. We take its filtering time as a *lower bound* of overall query response time, since the binary code from the authors reports only candidate sizes and filtering time.

11.7 Subgraph similarity searches

We compare the following algorithms for subgraph similarity search queries:

We randomly sampled as queries 100 graphs with $|V| \leq 20$ from AIDS and PROTEIN, respectively. In indexing phase, SGSimSearch takes a small amount of time and memory, for example, it takes 0.612 s and 389.3 kB memory for index

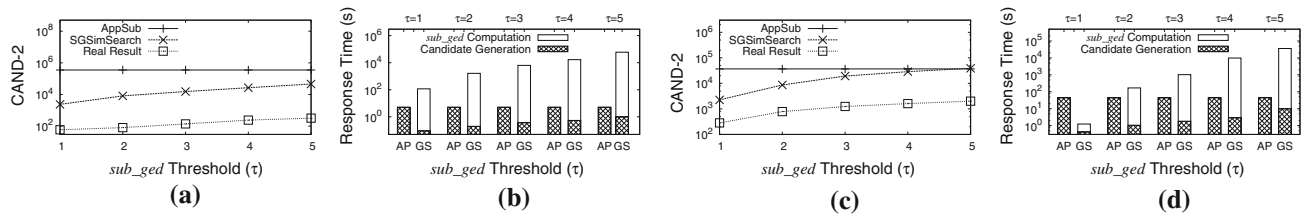

Fig. 17 Subgraph similarity search. **a** AIDS, Cand-2. **b** AIDS, query response time. **c** PROTEIN, Cand-2. **d** PROTEIN, query response time

on AIDS. Since AppSub does not involve an index, we compare Cand-2 and query response time in Fig. 17a–d.

As to Cand-2, AppSub exhibits a constant manner under smaller thresholds [38]. SGSimSearch delivers less Cand-2 than AppSub under small τ but shows a rising trend. Hence, the number of Cand-2 from SGSimSearch overtakes that of AppSub on PROTEIN when τ is as large as 5. Although SGSimSearch's total response time is more than AppSub's filtering time, AppSub's candidates need verification. In consideration of more candidates and lack of optimization on verification, it is unlikely that AppSub would outperform in terms of total response time under small τ settings.

11.8 Scalability evaluation

We evaluate the scalability of the algorithms against data graph size, data graph density, and dataset cardinality.

11.8.1 Varying data graph size

We evaluate the scalabilities of GSimSearch, κ-AT-Search, and SEGOS regarding data graph size on synthetic datasets. Five datasets with density 0.1 were generated. Each dataset has 4 k graphs, and the average graph size of the datasets ranges in {100, 200, 300, 400, 500}. Thus, the average number of vertices reaches 100 when $|E| = 500$. $q = 3$ was chosen for GSimSearch. Default parameter settings for the other algorithms were applied. We show the results under $\tau = 2$.

Figure 18a advises all algorithms have comparable indexing performance when graphs are small and take longer on larger graphs. κ-AT-Search scales with the smallest growth rate and is the most time-efficient when graphs are large. GSimSearch takes more time, 72.0x larger than κ-AT-Search, to build the index due to a larger number of q-grams in graphs when $|E| = 500$. Regarding index size in Fig. 18b, all algorithms need larger space to store the indexes for larger graphs, and q-gram-based approaches have smaller indexes than SEGOS under the given settings.

The response time for 100 queries are visualized in Fig. 18c. With respect to candidate generation, κ-AT-Search scales the best without notable change along with the increase in graph size, while the other two consume longer time gradually. More specifically, GSimSearch, having a small starting point, overtakes SEGOS when graphs are larger than 400. We argue that GSimSearch and SEGOS are, in general, more sensitive to graphs size than κ-AT-Search on indexing and filtering, but outperform κ-AT-Search on overall response time for fewer candidates (e.g., 1.8 and 5.5 k against 20.1 k when $|E| = 400$). Considering verification plays the major part in similarity queries, we nevertheless suggest spending reasonably more time on indexing and filtering so that, as a benefit, the total response time can be reduced.

11.8.2 Varying data graph density

We evaluate the scalability against graph density on synthetic datasets. We set the number of graphs to 4 k, the average graph size to 60, and varied the average density in {0.2, 0.4, 0.6, 0.8}. We fix τ to 2 and plot Cand-2 number and response time in Fig. 18d, e, respectively.

We observe that all algorithms take longer time to response according to the growth of density. As for candidate generation time, κ-AT-Search's is smaller when graph gets denser. This is due to the fact that denser graphs are more prone to be underflowing. In this case, less time is spent on candidate generation, with underflowing graphs immediately becoming candidates. Note we used the smallest q-gram size $q = 1$ for κ-AT-Search. GSimSearch spends more time on candidate generation with density, since there are more path-based q-grams to process. This is rewarding during online query processing phase, as demonstrated by the smallest candidate size. SEGOS does not display notable change in time consumption on candidate generation when graphs become denser. Since *ged* computation time exhibits remarkable growth on denser graphs, GSimSearch is the most overall time-efficient for its least candidate size.

11.8.3 Varying dataset cardinality

We evaluate the scalability against dataset cardinality under $\tau = 2$. We sampled 20–100 % graphs from AIDS without edge labels. Figure 18f compares the three join algorithms.

From the square root of total running time, we perceive the quadratic growth of all the three algorithms, given that the real join result has a quadratic growth in dataset cardinality.

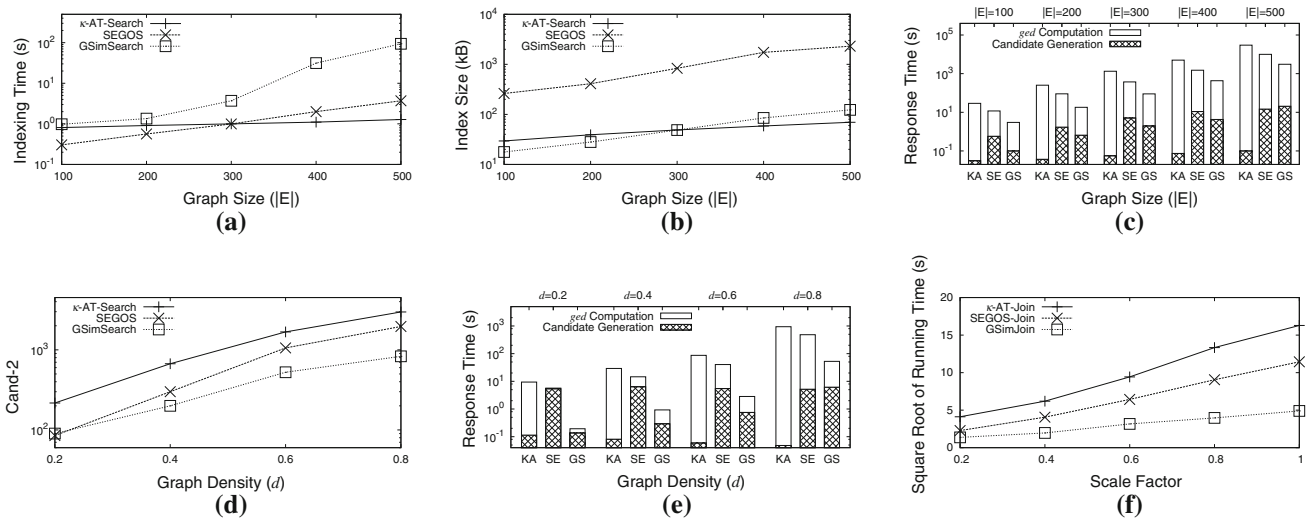

Fig. 18 Scalability evaluation. **a** Synthetic, indexing time. **b** Synthetic, index size. **c** Synthetic, query response time. **d** Synthetic, Cand-2. **f** Synthetic, query response time. **g** AIDS, total running time

The numbers of real join results are 7, 24, 44, 80, and 129 for the five scales, respectively. The GSimJoin algorithm demonstrates advantage over the others as its growth rate is smallest, with an overall speedup of as large as 22.1x against κ-AT-Join, and 5.5x against SEGOS-Join.

12 Related work

Similarity queries are important in various applications.

Graph similarity search Structure similarity search receives considerable attention lately. Closure-Tree is put forward to identify k graphs that are most nearly isomorphic to the query [12]. To formalize a general definition of structural similarity, graph edit distance is employed to measure the difference [38]. A recent advance is to employ κ-AT [27] as q-grams for edit distance-based similarity search. It builds inverted index by decomposing graphs into κ-AT's and perform filtering by comparing a count filtering-based distance lower bound with the threshold. The latest effort SEGOS [31] proposes an indexing and query processing framework for the same problem. GSimSearch belongs to this category.

Graph similarity containment search Subgraph similarity search is to retrieve graphs that approximately contain the query. Grafil [36] develops a feature-based pruning technique for subgraph similarity search, and similarity is defined as the number of missing edges with respect to maximum common subgraph. GrafD-index [22] exploits effective pruning and validation rules to tackle the problem of connected subgraph similarity search. As the counterpart, supergraph similarity search is also investigated to retrieve graphs that are approximately contained by the query [23].

Graph similarity matching Similarity queries on large graphs have also been studied. SAPPER [39] solves the problem of similarity all-matching, that is, to find all embeddings missing a given number of edges from the query in a large graph. TreeSpan [44] is the most up-to-date solution leveraging the query's spanning trees on demand. Neighborhood-based similarity [14] is also considered.

Graph edit distance computation Another line of related research focuses on graph edit distance computation. So far, the fastest exact solution is credited to an A*-based algorithm incorporating a bipartite heuristic [19]. To render it less computationally demanding, approximate methods are proposed to find suboptimal answer, for example, [9,18].

String and tree similarity query String similarity queries are well studied, and q-gram technique is widely applied, especially with edit distance constraints [10,33]. Others include (1) chunk-based approach, utilizing non-overlapping substrings [15,17,28,29]; (2) enumeration-based approach, enumerating resulting strings after editing [30]; and (3) trie-based approach, indexing strings in a tree structure [28,40].

q-gram-like structures, such as q-level binary branches [37], pq-grams [1], are also defined on tree-structured data. Parent–child and sibling relations are encoded in these substructures, which is not explicitly available in graphs. In addition, the lower bounds established through tree-based q-grams are usually loose for graphs due to the exponential coverage.

13 Conclusion

In this paper, we study three types of graph similarity queries with edit distance constraints. Unlike previous methods using trees or star structures, we propose a method exploiting the number of common fixed-length paths between pairs of graphs. Two filtering techniques are developed to handle both scattered and clustered edit operations as well as facilitate the

graph edit distance computation. Degree-associated structural information is also exploited to reduce candidate size and enhance runtime performance. Comprehensive experiments conducted on real and synthetic datasets demonstrate that the new algorithms outperform the existing methods based on either tree-based q-grams or star structures.

Acknowledgments X. Lin is supported by ARC DP0987557, DP110102937, DP120104168, NSFC61232006 and NSFC61021004. W. Wang is supported by ARC DP130103401 and DP130103405. C. Xiao and Y. Ishikawa are supported by FIRST Program, Japan.

References

1. Augsten, N., Böhlen, M., Gamper, J.: The pq-gram distance between ordered labeled trees. ACM Trans. Database Syst. (TODS) **35**(1), 1–36 (2010)
2. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval, 1st edn. Addison-Wesley Longman Publishing Co, Inc, Boston (1999)
3. Bunke, H., Allermann, G.: Inexact graph matching for structural pattern recognition. Pattern Recogn. Lett. **1**(4), 245–253 (1983)
4. Chaudhuri, S., Ganti, V., Kaushik, R.: A primitive operator for similarity joins in data cleaning. In: ICDE, p. 5 (2006)
5. Chen, C., Yan, X., Yu, P.S., Han, J., Zhang, D.-Q., Gu, X.: Towards graph containment search and indexing. In: VLDB, pp. 926–937 (2007)
6. Ciaccia, P., Patella, M., Zezula, P.: M-tree: An efficient access method for similarity search in metric spaces. In: VLDB, pp. 426–435 (1997)
7. Cottell, J.J., Link, J.O., Schroeder, S.D., Taylor, J., Tse, W.C., Vivian, R.W., Yang, Z.-Y.: Antiviral Compounds, patent WO2009005677 (2009)
8. Fagin, R., Lotem, A., Naor, M.: Optimal aggregation algorithms for middleware. In: PODS, pp. 102–113 (2001)
9. Fankhauser, S., Riesen, K., Bunke, H.: Speeding up graph edit distance computation through fast bipartite matching. In: GbRPR, pp. 102–111 (2011)
10. Gravano, L., Ipeirotis, P.G., Jagadish, H.V., Koudas, N., Muthukrishnan, S., Srivastava, D.: Approximate string joins in a database (almost) for free. In: VLDB, pp. 491–500 (2001)
11. Hart, P., Nilsson, N., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Syst. Sci. Cybern. **4**(2), 100–107 (1968)
12. He, H., Singh, A.K.: Closure-tree: an index structure for graph queries. In: ICDE, p. 38 (2006)
13. Justice, D., Hero, A.O.: A binary linear programming formulation of the graph edit distance. IEEE Trans. Pattern Anal. Mach. Intell. **28**(8), 1200–1214 (2006)
14. Khan, A., Li, N., Yan, X., Guan, Z., Chakraborty, S., Tao, S.: Neighborhood based fast graph search in large networks. In: SIGMOD Conference, pp. 901–912 (2011)
15. Li, G., Deng, D., Wang, J., Feng, J.: Pass-join: a partition-based method for similarity joins. PVLDB **5**(1), 253–264 (2012)
16. Pulleyblank, W.R.: Handbook of Combinatorics Chapter Matchings and Extensions, vol. 1. MIT Press, Cambridge (1995)
17. Qin, J., Wang, W., Lu, Y., Xiao, C., Lin, X.: Efficient exact edit similarity query processing with the asymmetric signature scheme. In: SIGMOD Conference, pp. 1033–1044 (2011)
18. Raveaux, R., Burie, J.-C., Ogier, J.-M.: A graph matching method and a graph matching distance based on subgraph assignments. Pattern Recogn. Lett. **31**(5), 394–406 (2010)
19. Riesen, K., Fankhauser, S., Bunke, H.: Speeding up graph edit distance computation with a bipartite heuristic. In: MLG (2007)
20. Robles-Kelly, A., Hancock, E.R.: Graph edit distance from spectral seriation. IEEE Trans. Pattern Anal. Mach. Intell. **27**(3), 365–378 (2005)
21. Sanfeliu, A., Fu, K.-S.: A distance measure between attributed relational graphs for pattern recognition. IEEE Trans. Syst. Man Cybern. **13**(3), 353–362 (1983)
22. Shang, H., Lin, X., Zhang, Y., Yu, J.X., Wang, W.: Connected substructure similarity search. In: SIGMOD Conference, pp. 903–914 (2010)
23. Shang, H., Zhu, K., Lin, X., Zhang, Y., Ichise, R.: Similarity search on supergraph containment. In: ICDE, pp. 637–648 (2010)
24. Silva, A., Jr, W.M., Zaki, M.J.: Mining attribute-structure correlated patterns in large attributed graphs. PVLDB **5**(5), 466–477 (2012)
25. Slavík, P.: A tight analysis of the greedy algorithm for set cover. In: STOC, pp. 435–441 (1996)
26. Tian, Y., Patel, J.M.: TALE: a tool for approximate large graph matching. In: ICDE, pp. 963–972 (2008)
27. Wang, G., Wang, B., Yang, X., Yu, G.: Efficiently indexing large sparse graphs for similarity search. IEEE Trans. Knowl. Data Eng. **24**(3), 440–451 (2012)
28. Wang, J., Li, G., Feng, J.: Trie-join: efficient trie-based string similarity joins with edit-distance constraints. PVLDB **3**(1), 1219–1230 (2010)
29. Wang, W., Qin, J., Chuan, X., Lin, X., Shen, H.T.: Vchunkjoin: an efficient algorithm for edit similarity joins. IEEE Trans. Knowl. Data Eng. **99** (preprints) (2012)
30. Wang, W., Xiao, C., Lin, X., Zhang, C.: Efficient approximate entity extraction with edit distance constraints. In: SIGMOD Conference, pp. 759–770 (2009)
31. Wang, X., Ding, X., Tung, A.K.H., Ying, S., Jin, H.: An efficient graph indexing method. In: ICDE, pp. 210–221 (2012)
32. Williams, D.W., Huan, J., Wang, W.: Graph database indexing using structured graph decomposition. In: ICDE, pp. 976–985 (2007)
33. Xiao, C., Wang, W., Lin, X.: Ed-Join: an efficient algorithm for similarity joins with edit distance constraints. PVLDB **1**(1), 933–944 (2008)
34. Yan, X., Han, J.: gSpan: Graph-based substructure pattern mining. In: ICDM, pp. 721–724 (2002)
35. Yan, X., Yu, P.S., Han, J.: Graph indexing: a frequent structure-based approach. In: SIGMOD Conference, pp. 335–346 (2004)
36. Yan, X., Yu, P.S., Han, J.: Substructure similarity search in graph databases. In: SIGMOD Conference, pp. 766–777 (2005)
37. Yang, R., Kalnis, P., Tung, A.K.H.: Similarity evaluation on tree-structured data. In: SIGMOD Conference, pp. 754–765 (2005)
38. Zeng, Z., Tung, A.K.H., Wang, J., Feng, J., Zhou, L.: Comparing stars: on approximating graph edit distance. PVLDB **2**(1), 25–36 (2009)
39. Zhang, S., Yang, J., Jin, W.: SAPPER: subgraph indexing and approximate matching in large graphs. PVLDB **3**(1), 1185–1194 (2010)
40. Zhang, Z., Hadjieleftheriou, M., Ooi, B.C., Srivastava, D.: Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In: SIGMOD Conference, pp. 915–926 (2010)
41. Zhao, P., Yu, J.X., Yu, P.S.: Graph indexing: Tree + delta >= graph. In: VLDB, pp. 938–949 (2007)
42. Zhao, X., Xiao, C., Lin, X., Wang, W.: Efficient graph similarity joins with edit distance constraints. In: ICDE, pp. 834–845 (2012)
43. Zhu, F., Qu, Q., Lo, D., Yan, X., Han, J., Yu, P.S.: Mining top-k large structural patterns in a massive network. PVLDB **4**(11), 807–818 (2011)
44. Zhu, G., Lin, X., Zhu, K., Zhang, W., Yu, J.X.: TreeSpan: efficiently computing similarity all-matching. In: SIGMOD Conference, pp. 529–540 (2012)