The VLDB Journal (2018) 27:53–78
https://doi.org/10.1007/s00778-017-0487-0

REGULAR PAPER

Efficient structure similarity searches: a partition-based approach

Xiang Zhao[1,2] · Chuan Xiao[3] · Xuemin Lin[4] · Wenjie Zhang[4] · Yang Wang[4]

Received: 19 February 2017 / Revised: 2 August 2017 / Accepted: 3 October 2017 / Published online: 24 October 2017

Abstract Graphs are widely used to model complex data in many applications, such as bioinformatics, chemistry, social networks, pattern recognition. A fundamental and critical query primitive is to efficiently search similar structures in a large collection of graphs. This article mainly studies threshold-based graph similarity search with edit distance constraints. Existing solutions to the problem utilize *fixed-size overlapping* substructures to generate candidates, and thus become susceptible to large vertex degrees and distance thresholds. In this article, we present a partition-based approach to tackle the problem. By dividing data graphs into *variable-size non-overlapping* partitions, the edit distance constraint is converted to a graph containment constraint for candidate generation. We develop efficient query processing algorithms based on the novel paradigm. Moreover, candidate-pruning techniques and an improved graph edit distance verification algorithm are developed to boost the performance. In addition, a cost-aware graph partitioning method is devised to optimize the index. Extending the partition-based filtering paradigm, we present a solution to the top-*k* graph similarity search problem, where tailored filtering, look-ahead and computation-sharing strategies are exploited. Using both public real-life and synthetic datasets, extensive experiments demonstrate that our approaches significantly outperform the baseline and its alternatives.

Keywords Graph database · Similarity query · Graph edit distance · Top-*k* search

Electronic supplementary material The online version of this article (doi:10.1007/s00778-017-0487-0) contains supplementary material, which is available to authorized users.

✉ Chuan Xiao
chuanx@nagoya-u.jp

1 National University of Defense Technology, Changsha, China

2 Collaborative Innovation Center of Geospatial Technology, Wuhan, China

3 Nagoya University, Nagoya, Japan

4 The University of New South Wales, Sydney, Australia

1 Introduction

Recent decades have witnessed a rapid proliferation of data modeled as graphs, such as biological interactions and business processes. As a fundamental and critical query primitive, graph search, which retrieves the occurrences of a query structure in a database, is frequently issued in these application domains, and hence attracts extensive attention. Due to the existence of data inconsistency, such as erroneous data entry, natural noise, and different data representations in different sources, a recent trend is to study similarity queries. A structure similarity search query finds data graphs from a graph collection that are similar to a query graph.

Thus far, various similarity or distance measures have been utilized to quantify the similarity between graphs, e.g., the measures based on maximum common subgraphs [26], or missing edges [31]. Among them, graph edit distance (GED) stands out for its elegant properties: (1) It is a metric applicable to all types of graphs, and (2) it captures precisely the structural difference (both vertex and edge) between graphs.[1] Thus, it finds a wide spectrum of applications of different domains, including object recognition in computer vision [3], and molecule analysis in chem-informa-tics [13]. For a notable example, *compound screening* in the process of drug development entails efficient structure similarity searches.

[1] An elaborated discussion is provided in Part A of supplementary material to this article.

The *structure–activity* relationship indicates that the biological activity of a compound is usually determined by its chemical structure. In light of it, for the investigation of a new chemical compound, chemists may query the existing massive chemical database with the compound, in order to find compounds with similar structures.

Driven by these reasons, we investigate structure similarity search with edit distance constraints in this research: Given a data graph collection and a query, we find all the data graphs whose GED to the query is within a threshold. However, the notorious GED computation (NP-hard [29]) poses serious algorithmic challenges. Therefore, state-of-the-art solutions are mainly based on a *filter–verify* strategy, which first generates a set of promising candidates under a looser constraint, and then verifies them with the expensive GED computation. Inspired by the q-gram idea for string similarity queries, the notions of tree-based q-gram [23] and path-based q-gram [33] were proposed. Both studies convert the distance constraint to a count-filtering condition, i.e., a requirement on the number of common q-grams, based on the observation that if the GED between two graphs is small, the majority of q-grams in one graph are preserved when transforming one to the other. Besides q-gram features, star structure [29] was also proposed, which is exactly the same as tree-based 1-gram. Rather than count common features, a method was developed to compute the lower and upper bounds of GED through bipartite matching between the star representations of two graphs [29]. The method was later equipped with a two-level index and a cascaded search strategy to find candidates [24]. Lately, branch structure was conceived through shrinking star structures by removing the leaf vertices [34], and then integrated with several mixed filters.

We summarize the aforementioned work, i.e., tree- and path-based q-grams, star and branch structures, as *fixed-size overlapping* substructure-based approaches, as the adopted features share two common characteristics: (1) *fixed size*—being trees of the same depth (tree-based q-grams and star structures) or paths of the same length (path-based q-grams) and *overlapping*—sharing vertices and/or edges in the original graphs such that one edit operation affects multiple substructures. Thus, these approaches inevitably suffer from the following drawbacks. (1) They do not take full advantage of the global topological structure of graphs and the distributions of data graphs/query workloads, and fixing substructure size limits its selectivity, being inelastic to databases and queries. (2) Redundancy exists among features, hence making their filtering conditions—all established in a pessimistic way to evaluate the effect of edit operations—vulne-rable to large vertex degrees or large distance thresholds. To overcome the shortcomings, we propose to leverage *variable-size non-overlapping* substructures via graph partitioning.

In this research, we present a novel solution following the filter–verify fashion (cf. Fig. 3), which is inspired by pigeonhole principle and instantiated by a refreshing filtering paradigm that divides data graphs into variable-size non-overlapping partitions. In contrast to fixed-size overlapping substructures, partition-based scheme is less prone to be affected by vertex degrees and can accommodate larger distance thresholds in practice with good selectivity, being adaptive to data and query graphs. This enables us to conduct similarity searches on a wider range of applications with larger thresholds. In case the *offline* partitioning of data graphs does not well fit the structural characteristics of the ever-changing queries, we explore the idea of dynamic partition filtering, and advanced *online* pruning strategies are accordingly proposed to reduce candidates. First, enhanced matching condition via neighborhood completion is enforced for matching partition with respect to the distance constraint. Second is to dynamically rearrange partitions to adapt to the online query by recycling and making use of the information in mismatching partitions. Furthermore in GED evaluation, we design a verification method by extending matching partitions to share the computation between filtering and verification. Additionally, a cost model is devised to compute quality partitioning of data graphs for a workload of queries.

Sometimes, it may be difficult to give a numerical distance threshold to the system, due to lack of an overall image of the database. One possible solution is to conduct top-k similarity searches; that is, to find the most similar k graphs to the query. However, handling top-k search with a threshold-based algorithm is rather expensive, since it has to enumerate the threshold incrementally, and execute the search procedure for each threshold. In response to this, we propose a top-k search algorithm with a hierarchical inverted index, which is among the first attempts of this style. On top of it, tailored pruning and delicate look-ahead strategies are availed to boost the performance.

Contribution This article is a substantial extension of our previous work [32]. We have made the following major updates: (1) In Sect. 4.2, we explore the idea of dynamic partitioning from a new perspective and propose enhanced matching condition that can be invoked on any matching partition under the partition-based filtering scheme. (2) In Sect. 4.3, we make a more smooth transition between the idea of recycling mismatching partitions and the algorithm by providing Theorem 3. It shows the optimization is NP-hard, and hence, a heuristic algorithm comes for efficient solution. (3) We introduce the concept of completeness of filtering methods, and show that the proposed filtering methods satisfy the completeness in Theorems 1, 2 and 4, respectively. An in-depth comparison among the filtering methods are also presented in Sect. 4.4. (4) In Sect. 5.2, we enhance the extension-based verification algorithm by removing the constraint of requiring only one matching partition. While retaining correctness, the generic version of the algorithm exhibits higher efficiency in terms of running time. (5) In

Sect. 7, we extend the partition-based filtering scheme to handle top-k graph similarity search. By level-wise merging the partition-based inverted index, we put forward an agglomerative index, on top of which a top-k search procedure with look-ahead and computation-sharing strategies are devised. (6) For empirical studies in Sect. 8, we conduct additional experiments on both real-life and synthetic datasets to verify the new techniques and methods. Moreover, the latest algorithms for graph similarity search and GED computation are involved for comprehensive evaluation. (7) In Sect. 9, we update the related work by incorporating some state-of-the-art research on top-k (sub-)graph exact and similarity search that has popped up most recently. (8) A brief extension of our solution to deal with *supergraph* similarity search queries is supplied in Part B of supplementary material to this article.

2 Preliminaries

This section provides the background knowledge.

2.1 Problem definition and notations

For ease of exposition, we focus on *simple* graphs; without loss of generality, our approaches can be extended to directed or multigraphs.

A graph g is represented in a triple (V_g, E_g, l_g), where V_g is a set of vertices, $E_g \subseteq V_g \times V_g$ is a set of edges, and l_g is a labeling function that assigns labels to vertices and edges. $|V_g|$ and $|E_g|$ are the number of vertices and edges in g, respectively. $l_g(v)$ denotes the label of vertex v, and $l_g(u, v)$ denotes the label of the edge (u, v). γ_g is the maximum vertex degree in g, and δ_g is the average vertex degree in g.

A *graph edit operation* is an edit operation to transform one graph to another [2,18], including:

- 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; and
- change the label of an edge.

The *graph edit distance* (GED) between graphs g and g', denoted by $GED(g, g')$, is the minimum number of edit operations that transform g to g'. The edit operations required by the transformation are referred as "edit error" in g' with respect to g. GED is a *metric*; nevertheless, computing GED between two graphs is NP-hard [29]. In this study, we use GED to capture the similarity between graphs, and two graphs are *similar* if their GED is not larger than a distance threshold τ. For brevity, we may use "distance" for "graph edit distance" when there is no ambiguity.

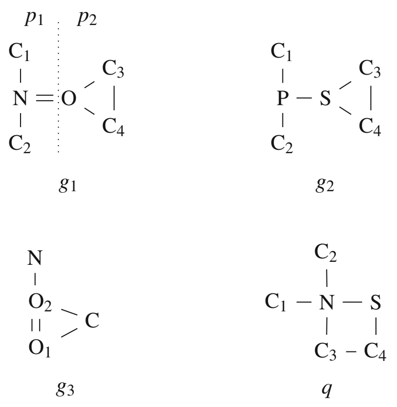


Fig. 1 Sample data graphs and query graphs

Example 1 In Fig. 1, there is a sample collection of data graphs $G = \{g_1, g_2, g_3\}$, where each graph models a molecule with vertex labels representing atoms and edges being chemical bonds. Subscripts are added to vertices with identical labels in the a single graph for differentiation, while they correspond to the same atom symbol; for example, $\mathtt{C}_1$ and $\mathtt{C}_2$ both refer to carbon atom. g_2 can be transformed to q by three edit operations: relabel $\mathtt{P}$ to $\mathtt{N}$, delete the edge between $\mathtt{S}$ and $\mathtt{C}_3$, and insert an edge between $\mathtt{N}$ and $\mathtt{C}_3$. So $GED(g_2, q) = 3$. Additionally, one may verify that $GED(g_2, q) < GED(g_1, q) < GED(g_3, q)$.

Problem 1 *(threshold-based graph similarity search) Given a data graph collection G, a query graph q, and an edit distance threshold τ, threshold-based similarity search finds all the data graphs $g \in G$ such that $GED(g, q) \leq \tau$.*

Problem 2 *(top-k graph similarity search) Given a data graph collection G, a query graph q, and an integer k, top-k similarity search finds a subset $R \subseteq G$, such that $|R| = k$ and $\forall g \in R, \nexists g' \in G \setminus R, GED(g', q) < GED(g, q)$.*

Example 2 Further to Example 1, assume that one inputs a query graph q. Threshold-based graph similarity search (Problem 1) with $\tau = 3$ returns graph g_2 as the answer, as only $GED(g_2, 1) \leq 3$; top-k graph similarity search (Problem 2) with $k = 2$ returns graphs g_2 and g_1 as the answers, since $GED(g_3, q) > GED(g_1, q) > GED(g_2, q)$.

For ease of reference, we list the major notations in Table 1. In the sequel, we first present a partition-based solution to Problem 1 with an inverted index and then extend to handle Problem 2 with a hierarchical index. We focus on solving the problems exactly on in-memory settings.

2.2 Prior work on Problem 1

Approaching Problem 1 with sequential scan is extremely costly, because one has to not only access the whole database

Table 1 Notations

Symbol	Description
g	A data graph
G	A collection of graphs
q	A query graph
Q	A query workload
v	A single vertex of a graph
$nb(v)$	The set of neighboring vertices of v
γ_g	The maximum vertex degree in g
p	A partition of a graph
P_g	A graph partitioning of g constituted of p's
$\hat{p}$	An extended partition of p
Δp	The extended portion of $\hat{p}$
seq_p	The revised QISequence of p
I	An inverted index of partitions
I_p	The postings list of I for entry p
$\mathcal{M}$	A lookup table from graph identifiers to Boolean
$\mathcal{P}$	The universe of index partitions
$\mathcal{P}_m$	The set of matching partitions

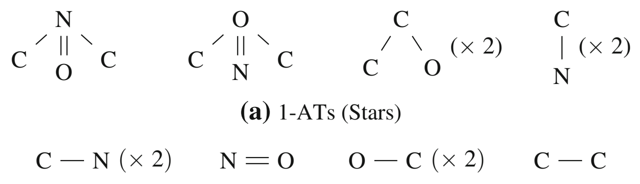
but also one by one conduct the NP-hard GED computations. Thus, the state-of-the-art solutions address the problem in a *filter–verify* fashion: first generate a set of candidates that satisfy necessary conditions of the edit distance constraints and then verify with edit distance computation.

Inspired by the q-gram concept in string similarity queries, *tree*-based q-grams [23] were first defined on graphs. For each vertex v, a κ-AT (or a q-gram) is a tree rooted at v with all vertices reachable in κ hops. A count-filtering condition on the minimum number of common κ-AT's between the data graph g and query graph q is established as

$$\max(|V_g| - \tau \cdot \Lambda(g),\ |V_q| - \tau \cdot \Lambda(q)),$$

where $\Lambda = 1 + \gamma \cdot \frac{(\gamma-1)^{\kappa}-1}{\gamma-2}$, γ is the maximum vertex degree of a graph, and κ is the a given number of hops. The lower bound tends to be small, and even below zero if there is a large-degree vertex in the graph and/or a large distance threshold, hence rendering it useful only on sparse graphs. To relieve the issue, *path*-based q-grams [33] were proposed, and techniques exploiting both matching and mismatching q-grams. Nonetheless, the exponential number of paths imposes a performance concern. Moreover, the inability to handle large vertex degree and distance threshold is inherited.

A *star* structure [29] is exactly a 1-gram defined by κ-AT. It employs a disparate philosophy for filtering based on bipartite matching between star structures of two graphs. Let $SED(g,\ q)$ denote the sum of pairwise distances from the bipartite matching of stars between g and q. A filtering condition is established on the upper bound of $SED(g,\ q)$



(a) 1-ATs (Stars)

(b) Path-based 1-grams

Fig. 2 Fixed-size substructures of g_1

$$\tau \cdot \max(4,\ 1 + \max(\gamma_g,\ \gamma_q)),$$

where γ_g (resp. γ_q) is the maximum vertex degree of graph g (resp. q), and the condition is also proportional to the maximum vertex degree. Based on star structures, a two-level index and a cascaded search strategy were devised [24]. While it is superior to star structure in search strategy, the basic filtering principle remains the same. Its performance is dependent on the parameters controlling the index access, whereas choosing appropriate parameter values is by no means an easy task. In addition, verification was not involved in the evaluation, and thus, the overall performance is not unveiled.

We summarize the aforementioned solutions as *fixed-size overlapping* substructure-based approaches. The major advantages of them are (1) substructures are easy to generate and manipulate, e.g., fast equality check, and (2) indexing space is pre-defined, e.g., trees or paths of fixed sizes. Intuitively, fewer candidates are usually associated with more selective features for filtering. Nevertheless, fixed-size features express little global structural information within the graphs and with respect to the whole database, and thus, feature selectivity is not well considered. In other words, it is difficult to balance the selectivities of frequent and infrequent features to achieve a collective goal on the number of candidates. Moreover, they are forced to accept the worst case assumption that edit operations occur at locations with the greatest feature coverage, i.e., modifying the most features. This effect is exacerbated by the overlap among features, and consequently, they are vulnerable to large vertex degrees and edit distance thresholds. The example below illustrates the aforementioned disadvantages.

Example 3 Consider the data graph g_1 and the query graph q in Fig. 1. Figure 2a shows the 1-ATs (or stars) of g_1, and in Fig. 2b are its path-based 1-grams. Assume $\tau = 2$. The count-filtering condition is $\max(6 - 2 \times 4, 6 - 2 \times 5) = -2$, while they share two 1-ATs. For path-based 1-grams, g_1 also satisfies the count-filtering condition. For star structures, bipartite matching on stars of g_1 and q returns $SED(g_1,\ q)$ as 4, while the allowed SED upper bound is $1 \cdot \max(4,\ (1+4)) = 5$, and thus cannot prune g_1. Therefore, all of them include g_1 as a candidate, whereas $GED(g_1, q) = 4 > \tau$.

Lately, there is another appealing method [34] that mixes the q-gram and mismatching disjoint partitions based ideas. For q-grams, it designs a *branch* structure which comprises a centering vertex and the edges incident to the vertex. Then, a bipartite matching cost is derived as the distance lower bound. For Example 3, the branch under vertex `N` can be represented as (`N`, {`-`, `-`, `=`}), and the (compact) branch filter produces a distance lower bound 1.5, less than $\tau = 2$. For mismatching disjoint partitions, it online partitions the query graph, and accumulates all the mismatches, the number of which is a distance lower bound. As to Example 3, it gives a distance lower bound 2, no larger than τ. Furthermore, a hybrid filter is conceived on the basis of the two lower bounds. Note that this algorithm can still be roughly categorized into the fixed-size substructure-based approaches, because (1) branches are akin to stars, and (2) *most* of the mismatching partitions are restricted up to size-3 (e.g., `-C-` and `C-C`) in practice. We make a thorough comparison with the state-of-the-art algorithms in Sect. 8.6.

2.3 Solution overview

In this research, the proposed partition-based solution is coined as Pars (partition-based similarity search), comprising two major components for indexing and query processing, respectively, as overviewed in Fig. 3. Indexing is usually done in an offline mode on a collection of data graphs, which is composed of two steps—data graph partitioning and inverted index construction. When a query comes for processing, it is handled in two phases—filtering and verification. For filtering, we first employ the basic partition-based filtering scheme to obtain a initial set of candidates, and then dynamic partition filtering scheme is availed to reduce candidates. The later is explored from two perspectives, namely enhanced matching condition and recycling mismatching partitions. Surviving candidates are eventually fed to an extension-based verification, which produces the final answers.

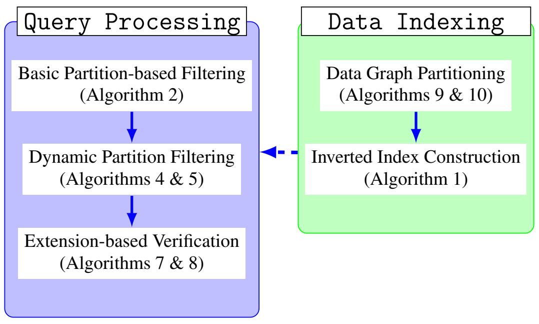

Query Processing
Basic Partition-based Filtering (Algorithm 2)
Dynamic Partition Filtering (Algorithms 4 & 5)
Extension-based Verification (Algorithms 7 & 8)
Data Indexing
Data Graph Partitioning (Algorithms 9 & 10)
Inverted Index Construction (Algorithm 1)

Fig. 3 Solution overview of Pars

3 A partition-based algorithm

In this section, we propose our partition-based algorithm for threshold-based structure similarity search, and start with an overview of the whole solution.

3.1 Partition-based filtering scheme

We illustrate the idea of partition-based filtering by an example and formalize the scheme afterward.

Example 4 Consider in Fig. 1 the graphs g_1 and q, and assume $\tau = 1$. g_1 is partitioned into non-overlapping p_1 and p_2, and neither partitions is contained by q. That is, at least one edit operation for each partition is necessary to transform g_1 to q. Thus, $GED(g_1, q)$ is at least 2, and hence, g_1 cannot satisfy the query constraint regarding q.

The example shows the possibility of filtering data graphs by partitioning them and carrying out a containment test against the query graph. Assume each data graph g is partitioned into $\tau + 1$ non-overlapping partitions. From the *pigeonhole principle*, $GED(g, q)$ must exceed τ if none of the $\tau + 1$ partitions is contained by q. Before formally presenting the filtering principle, we start with the concept of *half-edge graph* for defining data graph partitions.

Definition 1 (*Half-edge*) A *half-edge* is an edge with only one end vertex, denoted by $(u, \cdot)$, where u is the vertex that the edge is incident to.

Definition 2 (*Half-edge graph*) A *half-edge graph* g is a labeled graph, denoted by a tuple (V_g, E_g, l_g), where V_g is a set of vertices, $E_g \subseteq V_g \times V_g \cup V_g \times \{\cdot\}$ comprises a set of edges and a multiset of half-edges, and l_g is a labeling function that assigns labels to vertices and (half-) edges.

Definition 3 (*Half-edge subgraph isomorphism*) A half-edge graph g is *subgraph isomorphic* to a *graph* g', denoted as $g \sqsubseteq g'$, if there exists an *injection* $f : V_g \rightarrow V_{g'}$ such that

- $\forall u \in V_g, f(u) \in V_{g'} \wedge l_g(u) = l_{g'}(f(u))$;
- $\forall (u, v) \in E_g, (f(u), f(v)) \in E_{g'} \wedge l_g(u, v) = l_{g'}(f(u), f(v))$;
- $\forall (u, \cdot), (v, \cdot) \in E_g, \exists w, w' \in V_{g'} \setminus f(V_g), (f(u), w) \in E_{g'} \wedge (f(v), w') \in E_{g'} \wedge l_g(u, \cdot) = l_{g'}(f(u), w) \wedge l_g(v, \cdot) = l_{g'}(f(v), w')$, and
  - if $u = v$, $w \neq w' \in nb(f(u))$;
  - if $u \neq v$, $w \in nb(f(u))$, $w' \in nb(f(v))$,

  where $nb(v)$ is the set of neighbor vertices of v in g.

The major difference between Definition 3 and classic subgraph isomorphism lies in the third item for mapping half-edges. The last two conditions ensure that two half-edges of g do not map to an identical edge of g'.

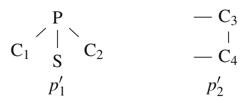


Fig. 4 Example of partitioning of g_2 in Fig. 1

If $g \sqsubseteq g'$, we say g is a *half-edge subgraph* of g', or g is *half-edge contained* by g'. Immediate is that half-edge subgraph isomorphism test is at least as hard as subgraph isomorphism test (NP-complete [5]). We may omit prefix "half-edge" onward when context is clear.

Definition 4 (*Graph partitioning*) A *partitioning* of a graph g is a division of the elements—vertices V_g and edges E_g—into collectively exhaustive and mutually exclusive non-empty groups with respect to V_g and E_g, i.e.,

$$P(g) = \{\, p_i \mid \cup_i p_i = V_g \cup E_g \wedge p_i \cap p_j = \emptyset,\ \forall i \neq j \,\},$$

where each p_i is a half-edge graph, called a *partition* of g.[2]

Example 5 Consider graph g_2 in Fig. 1. Figure 4 depicts one partitioning $P(g_2) = \{\, p'_1,\ p'_2 \,\}$ among many others, where p'_1 and p'_2 are two half-edge graphs.

In this research, we assume that an edge always exists with the end vertex that it is incident to; in other words, if two vertices u and v both are in partition p, the induced edge (u, v) is also in p. We abuse the notation $g \setminus p$ to notate the remaining subgraph of g except p. Furthermore, we call a partition p a *matching* partition if p is subgraph isomorphic to the query graph q, or equivalently, it *matches* the query graph; otherwise, p is a *mismatching* partition, or it *mismatches* the query graph.

Next, we state our partition-based filtering principle.

Lemma 1 *Consider a query graph q and a data graph g with a partitioning $P(g)$ of $\tau + 1$ partitions. If $GED(g, q) \leq \tau$, at least one of the $\tau + 1$ partitions matches q.*

Proof We prove by contradiction, and assume that none of the $\tau + 1$ partitions of $P(g)$ is subgraph isomorphic to q. By the definition of graph edit distance, we need to establish a one-to-one mapping f between V_g and V_q via *graph isomorphism*, in which case every $p \in P(g)$ is modified to be subgraph isomorphic to q. Recall that the partitions do not overlap with each other on any vertex or edge of g. Since none of p's are subgraph isomorphic to q, at least one edit operation is required to transform p to make it contained by q. Thus, $\tau + 1$ edit operations in total are essential to establishing the isomorphic mapping.

On the other hand, however, $GED(g, q) = \tau' \leq \tau$ suggests that only τ' edit operations are indispensable to fulfill the necessary transformation, in which case we can affect at most τ' partitions of $P(g)$ with $\tau + 1 - \tau' > 0$ partitions unaffected. This contradicts the assumption that none of the $\tau + 1$ partitions of $P(g)$ is contained by q. □

[2] A partition can be either connected or disconnected.

Algorithm 1: ParsIndex $(G,\ \tau)$
```
Input  : G is a collection of data graphs; τ is an edit distance
         threshold.
Output : An inverted index I.
1 $I \leftarrow \emptyset$;
2 foreach $g \in G$ do
3     $P_g \leftarrow$ GraphPartition $(g)$;
4     foreach partition $p \in P_g$ do
5         $I_p \leftarrow I_p \cup \{g\}$; /* add to postings list */
6 return $I$;
```

Corollary 1 *Consider a query graph q, a data graph g and a partitioning $P(g)$ of $\tau + 1$ partitions. If $GED(g, q) = \tau' \leq \tau$, at least $\tau + 1 - \tau'$ partitions match q.*

Proof We prove by contradiction, and assume there are only $\tau - \tau'$ partitions matching q. In this case, there are $\tau' + 1$ mismatching partitions of g against q. As the partitions are disjoint, each mismatching partition incurs at least one edit error. As a consequence, there needs in total at least $\tau' + 1$ edit operations to transform g to q. This implies that the distance between g and q is at least $\tau' + 1$, which contradicts with $GED(g, q) \leq \tau'$. Therefore, the corollary follows. □

Due to Corollary 1, we are able to build an index offline with a pre-defined $\tau_{\max}$, which works for all thresholds $\tau \leq \tau_{\max}$. From now on, we will focus on the $\tau = \tau_{\max}$ case, and refer to this as *basic partition*-based filtering scheme.

3.2 Similarity search framework

In light of Theorem 1, we propose a partition-based similarity search framework Pars for Problem 1. It encompasses two stages—indexing (Algorithm 1) and query processing (Algorithm 2). For Algorithm 1 (usually done offline), it takes as input a graph database G and a distance threshold τ, and constructs an inverted index. For each data graph g, it first divides g into $\tau + 1$ partitions by calling GraphPartition (Line 3, to be introduced in Sect. 6). Then, for each partition, it inserts the identifier of g into the postings list of that partition (Line 5). This completes the indexing stage.

In the online query processing stage, Algorithm 2 receives a query graph q, and probes the inverted index for candidate generation. We utilize a map to indicate the states of data graphs, which can be **uninitialized**, **true** or **false**. At first, the states are set to **uninitialized** for all data graphs (Line 1). Then, for each partition p in the inverted list, it tests whether p is contained by the query (Line 3). If so,

Algorithm 2: ParsQuery $(q,\ \tau,\ I)$

```
Input  : $q$ is a query graph; $I$ is an inverted index built on $G$;
         $\tau$ is an edit distance threshold.
Output : $R = \{g \mid GED(g,q) \le \tau, g \in G\}$.
1  $\mathcal{M} \leftarrow$ empty lookup table from graph identifier to boolean;
2  foreach $p$ in $I$ do
3      if SubgraphIsomorphism $(p, q, \emptyset)$ then
4          foreach $g$ in $I_p$ such that $\mathcal{M}[g]$ is uninitialized do
5              if SizeFilter $(g, q) \wedge$ LabelFilter $(g, q)$ then
6                  $\mathcal{M}[g] \leftarrow$ true;
7              else $\mathcal{M}[g] \leftarrow$ false;
8  foreach $g \in G$ such that $\mathcal{M}[g] =$ true do
9      if GEDVerification $(g, q) \le \tau$ then $R \leftarrow R \cup \{g\}$;
10 return $R$;
```

for each data graph with an **uninitialized** state in the postings list of p, it examines the graph through *size filtering* and *label filtering*. Size filtering tests whether the difference exceeds τ between the data and the query graph in terms of vertex and edge numbers. Label filtering examines whether the numbers of vertex and edge relabeling is more than τ, regardless of structures. The states of the qualified graphs are set to **true** and become candidates, while the states of the disqualified are set to **false** and will not be tested in the future (Lines 4–7). Finally, candidates are sent to GEDVerification, and results are returned in R (Line 9).

To ensure that Algorithm 2 finds all the answers to query q, we require the filtering method to satisfy the completeness. We first give a formal definition of completeness of a filtering method, and then, present Theorem 1.

Definition 5 (*Completeness*) Given a candidate pair (g, q), a filtering method is complete, provided that if (g, q) is similar, (g, q) must pass the filter.

Theorem 1 *The basic partition-based filtering scheme satisfies the completeness.*

Proof Assume the distance threshold is τ. Given a similar graph pair (g, q) of $GED(g, q) = \tau' \le \tau$. According to Lemma 1, by dividing g into $\tau + 1$ partitions, τ' edit operations affect at most τ' partitions, leaving $\tau + 1 - \tau'$ partitions unaffected. Each of the latter matches q and makes g pass the basic filter. Therefore, the completeness of the basic partition-based filtering scheme is satisfied. □

3.3 Cost analysis

In the query processing stage, the major concern is the response time, including filtering and verification time. Let $\mathcal{P}$ denote the universe of indexed partitions, each associated with a list of graphs having the partition. We model the overall cost of processing a query by

$$|\mathcal{P}| \cdot t_s + |\mathcal{P}_m| \cdot t_c + |C_q| \cdot t_d,$$

where (1) t_s is the average running time of a subgraph isomorphism test, which is conducted for every partition in $\mathcal{P}$; (2) t_c is the average time of retrieving (and merging) the postings lists of a matching partitions, which is carried out for $|\mathcal{P}_m|$ times, where $\mathcal{P}_m$ is the set of matching partitions; and (3) t_d is the average time of a GED computation, which is applied for $|C_q|$ times in total on the candidate set C_q.

Since the postings lists are usually short due to judicious graph partitioning (to be discussed in Sect. 6), subgraph isomorphism tests and GED computations play the major roles. Thanks to recent advances, subgraph isomorphism test can be done efficiently on small graphs and even large sparse graphs (with hundreds of distinct labels and up to millions of vertices) [1,22]. Our empirical study also demonstrates that subgraph isomorphism test is on average three orders of magnitude faster than GED computation. Moreover, t_c is much smaller than t_s, and $\mathcal{P}_m$ is always a subset of $\mathcal{P}$. Therefore, we argue that the major factor of the overall cost lies in GED computation, i.e., $|C_q| \cdot t_d$, and the key to improve system response time is to minimize the candidate set C_q.

It has been observed that the filtering performance of algorithms relying on *inclusive logic* over inverted index is determined by the selectivity of indexed features. A matching feature[3] is apt to produce many candidates if its postings list is long, i.e., it frequently appears in data graphs. Fixed-size features are generated irrespectively of frequency, and hence selectivity, while variable-size partitions offer more flexibility in constructing feature-based inverted index. We are able to choose the features reflecting the global structural information within data graphs and whole database, and obtain statistically more selective features. Further, partition-based features distinguish from those utilized by existing approaches in that partitions are non-overlapping. This property restricts that an edit operation can affect at most one feature, and thus, the number of features hit by τ edit operations is drastically reduced. As a result, unlike previous approaches, partition-based algorithms suffer little from the drawback of loose lower bounds when handling large thresholds and data and/or query graphs with large-degree vertices.

Before delving into the details of graph partitioning algorithms (Sect. 6), we first exploit two further optimizations to reduce candidates on top of the basic partition-based filtering scheme (Sect. 4), and also discuss efficient verification of candidates (Sect. 5).

[3] For example, a partition contained by the query for Pars, or a q-gram appearing in the query's q-gram multiset for κ-AT.

4 Dynamic partition filtering

We start with an illustrating example.

Example 6 Consider in Fig. 1 the data graph g_2 and query q, and $\tau = 1$. Assume we have partitioned g_2 to p'_1 and p'_2 as in Fig. 4. p'_1 is not contained by q but p'_2 is, making g_2 a candidate. However, if we adjust the partitioning by moving vertex S from p'_1 to p'_2, neither partitions will be contained by q, hence disqualifying g_2 being a candidate.

This example evidences the chance of adjusting the partitions according to an online query so that the pruning power of partition-based filtering is enhanced. In light of it, this section conceives a novel filtering technique, namely, *dynamic partition filtering scheme*, to exploit the observation. We concretize the idea from two different perspectives—enhanced matching condition and mismatching partition recycle, and integrate them into the tests of data graph partitions. In essence, the former is lightweight with certain restriction, while the latter is generic and widely applicable.

Next, we first adapt a graph encoding technique for efficient half-edge subgraph isomorphism test, based on which dynamic partition filtering will be presented.

4.1 Half-edge subgraph isomorphism test

QISequence [20] is a graph encoding technique originally proposed for efficient (non-half-edge) subgraph isomorphism test. We adapt and extend it to support *half-edges* and *disconnected* cases. The *revised* QISequence of a partition p is a *regular expression* $[[v_i e^*_{ij}]^{|V_p|}]$, which consists of constants, which denotes sets of vertices and edges, and operator symbols, which denote operations over these sets. When p is connected, seq_p is encoded based on a spanning tree of p. For constants, v_i is the ith vertex in the order of the spanning tree; for all $i > j$, e_{ij} encodes

- sEdge—the *spanning edge* between v_i and v_j in the spanning tree;
- bEdge—the *backward edges* between v_i and v_j in p but not in the spanning tree; and
- hEdge—the *half-edges* incident to v_i.

For operator symbols, a Kleene star e^*_{ij} denotes the smallest superset of e_{ij}, and a concatenation $v_i e^*_{ij}$ denotes the set of constants can be obtained by concatenating v_i and some e_{ij}'s; a exponentiation with a positive integer exponent $|V_p|$ denotes repeated concatenation of the base set of constants by $|V_p|$ times. For disconnected case with multiple connected components, sequences are generated for each component and then concatenated as QISequence.

To generate the QISequence of p, we start at the root of a spanning tree of p, and vertices of p are appended in the order of spanning tree traversal, each time a spanning edge with (possible) backward edges and half-edges.

Algorithm 3: BasicSubgraphIso $(p, q, \mathcal{F})$

```
Input  : p is a partition; q is a query graph; F is a mapping vector.
Output : A boolean indicating whether p ⊑ q.
1 if $|\mathcal{F}| = |V_p|$ then return true ;
2 $v \leftarrow$ next vertex in $seq_p$;
3 $U \leftarrow \{u \mid u \in$ FindCandidate$(v, seq_p, q, \mathcal{F})\}$;
4 foreach $u \in U$ do
5     $\mathcal{F}' \leftarrow \mathcal{F} \cup \{v \to u\}$;
6     if BasicSubgraphIso $(p, q, \mathcal{F}')$ then return true ;
7 return false;
```

Example 7 Consider the partition p'_1 in Fig. 4. Based on a spanning tree rooted at P, the sequence $seq_{p'_1}$ of p'_1 is shown in the left of Fig. 5, where solid lines represent spanning edges and half-edges (without the other end vertex), and dashed lines represent backward edges.

Algorithm 3 tests whether a partition p is subgraph isomorphic to the query q. It maps the vertices of p one after another, following the order of the QISequence of p to find a vertex mapping $\mathcal{F}$ from p to q via a depth-first search. For the current vertex v of p, if $seq_p[v]$ is the first term with sEdge $= nil$, it finds candidate vertices from all unmapped vertices in q; otherwise, it utilizes $seq_p[v]$.sEdge to shrink the search space. Candidate vertices are further checked by label ($l_p(v)$), backward edge ($seq_p[v]$.bEdge) and half-edge ($seq_p[v]$.hEdge) constraints successively. These are realized by FindCandidate (Line 3), which discovers valid candidate mappings of v in q by comparing vertex label and associated edges regarding the threshold. Then, we map v to one of the qualified vertices, and proceed with the next vertex. We call $\mathcal{F}$ a *partial mapping* if $|\mathcal{F}| < |V_p|$, or a *full mapping* if $|\mathcal{F}| = |V_p|$. If the current mapping cannot be extended to a full mapping, it backtracks to the precedent vertex in QISequence, and tries another mapping. The algorithm terminates when a full mapping is found, indicating p is subgraph isomorphic to q; or it fails to find any full mapping.

Complexity analysis It can be verified that if there exits a half-edge subgraph isomorphism from p to q, Algorithm 3

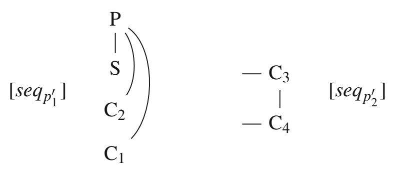


Fig. 5 Example of QISequences (solid lines—spanning and half-edges; dashed lines—backward edges)

must find it. The worst case time complexity remains the same as classic subgraph isomorphism test, i.e., $O(\gamma_q^{|V_p|})$.

4.2 Enhancing matching condition

Recall in Algorithm 2 that once a partition matches, the corresponding data graphs become candidates; only when none of partitions over a data graph match, the data graph will be pruned. As a consequence, we seek every opportunity to reduce the chance of matching partitions for fewer candidates. The following example manifests the possibility by taking into consideration the missing end vertices as well as their incident edges of matching partitions.

Example 8 Consider the graphs g and q in Fig. 6, and $\tau = 2$. g is partitioned into 3 half-edge graphs: $\mathtt{C}_1 - \mathtt{N}$, $> \mathtt{C}_2 - \mathtt{C}_3 -$, and $\mathtt{O} - \mathtt{C}_4$; q is partitioned into 3 half-edge graphs: $\mathtt{C}_1 = \mathtt{O}$, $>\mathtt{C}\ _2 - \mathtt{C}_3-$, and $\mathtt{C}_4 - \mathtt{Cl}$. Let p (in red) denote the second partition of g. p finds an isomorphism $\mathcal{F}$ (in blue) in q according to Algorithm 3, hence making g a candidate. Next, we try to expand p by growing end vertices $\mathtt{C}_1$, $\mathtt{N}$ and $\mathtt{O}$. If we investigate the (vertex-) induced subgraph $\hat{p}$, and its counterpart in q (circled by dashed lines), it takes 3 edit operations to make them identical, which is greater than τ. This portends that $\mathcal{F}$ is invalid with respect to τ. As there are no other isomorphisms from p to q, p should not match q. Further, it is also the case for the other two partitions, and thus, (g, q) is determined to be dissimilar.

The example depicts the scenario that even if a partition finds an isomorphism in the query graph, the mapping may be invalid, and we can disqualify the "matching" partition (after testing all isomorphic mappings) and carry on to examine other partitions. In this connection, we propose *enhanced matching condition* that limits the location of a partition in the query graph, and if all the mappings of the partition are determined to be invalid, it cannot match the query graph. This procedure eventually reduces the chance for a data graph to be a candidate. Following formally defines the concept of extended partition.

Definition 6 (*Extended partition*) Given a half-edge subgraph p of graph g, the extended partition of p, denoted by $\hat{p}$, is constructed by following steps:

(1) for each half-edge, grow the corresponding end vertex by $V_g \setminus V_p$, and include it in $V_{\Delta p}$;

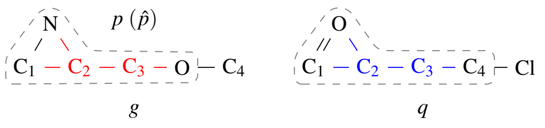


Fig. 6 Example of enhanced matching condition

(2) for each half-edge, remove it from E_p, and add the corresponding (normal) edge containing it into $E_{\Delta p}$; and
(3) for each vertex in $V_{\Delta p}$, restore the induced edges involving it in $E_g \setminus E_p$ if any, and include them in $E_{\Delta p}$.

An extended partition $\hat{p}$ is a graph such that $V_{\hat{p}} = V_p \cup V_{\Delta p}$ and $E_{\hat{p}} = E_p \cup E_{\Delta p}$. It can be seen that only steps (1) and (3) add elements to p. The basic idea of enhanced matching condition via extended partitions is to increase the chance of disqualifying a "matching" partition, and a data graph eventually, by limiting the location of p in q leveraging Δp. That is, based on a given mapping $\mathcal{F} : p \to q$, we are to find the mapping from $\hat{p}$ to q incurring the minimum edit errors, and if this number exceeds τ, $\mathcal{F}$ is invalid.

Nonetheless, it brings to our attention that there exists a combinatorial optimization; in other terms, although $\mathcal{F}$ is given and fixed, to find the best mappings for Δp is a non-trivial task, if there are multiple vertices in $V_{\Delta p}$ to map and settle. For the sake of elegant online performance, we resort to a lower bound of the minimum number of edit errors by utilizing the 1-hop neighborhoods of the vertices in $V_{\Delta p}$. Particularly, for each vertex $v \in V_{\Delta p}$, we select a mapping candidate from the *unused* vertex set $V_q \setminus V_{\mathcal{F}(p)}$ such that the connections to vertices in V_p matches. This is required by the isomorphism $\mathcal{F}$. Then, the number of mismatches on vertex label and connections to vertices in $V_{\Delta p}$ is logged. Among all possible mapping candidates, the minimum number of mismatches is kept to surrogate the number of edit errors at v. Adding together for all vertices in $V_{\Delta p}$ provides an estimation of real number of minimum edit errors. Since the mapping procedure does not consider the combination of the mapping candidates of $V_{\Delta p}$, the solution may not be globally feasible, and thus, renders a lower bound.

The aforementioned estimation provides the *first* portion of the overall filtering condition for pruning. Besides, we also take into account in enhanced matching condition the possible edit errors incurred by comparing the induced subgraph of $\mathcal{F}(p)$ with p. Recall that when matching p to q by Algorithm 3, we only examine the elements of p, in order to establish a subgraph isomorphism. Nevertheless, our ultimate goal is to find a transformation from g to q conforming the constraint based on *graph isomorphism*. The procedure of matching p to q overlooks the potential mismatches in q compared with p. Thus, we integrate the check of such mismatches, which together with the first portion makes up the final condition, as detailed in Algorithm 4.

Algorithm 4 takes as input a partition p, a query graph q, as well as a subgraph isomorphic mapping $\mathcal{F} : p \to q$, and determines whether $\mathcal{F}$ is valid according to enhanced matching condition. Specifically, we first construct the extended partition of p, and initialize a variable ε to 0, which accumulates and estimates the total number of edit errors (Lines 1 and 2). Then, we compute an estimation of edit errors to

Algorithm 4: EnhancedMatching $(p, q, \mathcal{F})$

```
Input  : $p$ is a partition; $q$ is a query graph; $\mathcal{F}$ is a mapping vector.
Output : A boolean indicating whether $\mathcal{F}$ is valid regarding $\tau$.
1  construct the extended partition $\hat{p}$ of $p$;
2  $\varepsilon \leftarrow 0$;                /* aggregate edit errors */
3  foreach vertex $v \in V_{\Delta p}$ do
4      $\rho \leftarrow \tau + 1$;
5      foreach vertex $u \in V_q \setminus V_{\mathcal{F}(p)}$ do
6          if all edges $(v, v') \in E_p$ matches $(u, \mathcal{F}(v'))$ then
7              if $l_g(v) = l_q(u)$ then $\sigma \leftarrow 0$ else $\sigma \leftarrow 1$;
8              foreach edge $(v, v') \in E_{\Delta p}$ do
9                  if $\nexists(u, u')$ such that
                   $u' \in V_q \setminus V_{\mathcal{F}(p)} \wedge l_g(v, v') = l_q(u, u')$ then
10                     $\sigma \leftarrow \sigma + 1$;
11             if $\sigma < \rho$ then $\rho \leftarrow \sigma$;
12     if $\rho \leq \tau$ then $\varepsilon \leftarrow \varepsilon + \rho$ else return false;
13 if $\varepsilon > \tau$ then return false;
14 foreach vertex $u \in V_{\mathcal{F}(p)}$ as per order in $\mathcal{F}$ do
15     foreach edge $(u, u')$ such that $u'$ is ahead of $u$ in $\mathcal{F}$ do
16         if $(\mathcal{F}^{-1}(u), \mathcal{F}^{-1}(u')) \notin E_p$ then $\varepsilon \leftarrow \varepsilon + 1$;
17 return $\varepsilon \leq \tau$;
```

match $\hat{p}$ to q, by selecting for each newly grown vertex in $V_{\Delta p}$ an unmapped vertex of q satisfying the mapping constraint. Foremost, the vertex has to fulfill the mapping constraints imposed by $\mathcal{F}$; otherwise, it cannot make a candidate (Line 6). Afterward, we examine the vertex labels and neighborhoods to lower bound the number of edit errors at this vertex, where only the minimum value is kept in ρ among all choices (Lines 7–11). If ρ is no more than τ, it is aggregated to ε; otherwise, the current vertex cannot find a counterpart conforming τ, and the mapping is immediately determined to be invalid (Line 12). Estimations for all vertices are then aggregated to ε, and compared with τ. If $\mathcal{F}$ survives through the first phase, we proceed to the second phase—to test the induced edges by $\mathcal{F}(p)$ against p, in which each mismatch contributes 1 edit error to ε (Lines 14–16). That is, if there is an edge (u, u') in the induced subgraph of $\mathcal{F}(p)$ but having no counterpart $(\mathcal{F}^{-1}(u), \mathcal{F}^{-1}(u'))$, it necessitates an edit operation. Again, all errors are aggregated to the existing ε, and compared with τ. If ε is not larger than τ, $\mathcal{F}$ is valid by enhanced matching condition; otherwise, it is determined to be invalid.

Complexity analysis In the first phase, there are at most $|E_p|$ newly grown vertices when all edges in p are half-edges. For each vertex, it computes the number of edit errors by exploring and comparing the neighborhoods of $|V_q|$ candidate vertices, each in $O(\gamma_g)$ time. Thus, the time complexity of the first phase is $O(\gamma_g |E_p||V_q|)$. Similarly, the second phase takes $O(\gamma_q |V_p|)$. In short, checking enhanced matching condition by Algorithm 4 is in $O(\gamma_g |E_p||V_q| + \gamma_q |V_p|)$.

Lemma 2 *Consider a matching partition p of a data graph g to a query graph q via mapping $\mathcal{F}$, and ε as derived by Algorithm 4. If $\mathcal{F}$ is a valid mapping regarding distance constraint τ, $\varepsilon \leq \tau$.*

Proof Denote by ζ the minimum number of edit operations to transform the extended partition to an induced subgraph of q after fixing $\mathcal{F}$, which is expected by the transformation between g and q. If $\mathcal{F}$ is valid, based on which g finds a feasible transformation to q, ζ must be no larger than τ. Hence, it is left to show that the derived ε is a lower bound of ζ.

ε is constituted of two portions—the *first* portion comes from the mismatches when mapping the extended part of p, and the second portion counts the overlooked mismatches on $\mathcal{F}$ when matching p against q. For the first portion, the neighborhood of every extended vertex in Δp is examined, and the estimation is made by matching it to an arbitrary vertex that is not used by $\mathcal{F}(p)$. In the optimal case, such choice cannot be arbitrary, and combinations of the candidate vertices need to be explored in order to determine the optimal edit cost. Thus, summing up the individual edit errors never overestimates the optimal cost. As to the *second* portion, it is the exact cost for the induced edges that are not currently included in $\mathcal{F}(p)$, which were not considered when matching p against q but need to be handled when transforming g to q. Consequently, the two portions together provides a lower bound of ζ, and thus the lemma is proved. □

Through the proof of Lemma 2, we see that although $\mathcal{F}$ is not a valid mapping, p may still match q under other mappings, based on which g finds a transformation to q conforming τ. Thus, we need to test *all* possible mappings of p in q. To integrate enhanced matching condition in the half-edge subgraph isomorphism test, we replace Line 6 of Algorithm 3 with "**if** BasicSubgraphIso $(p, q, \mathcal{F}) \wedge$ EnhancedMatching $(p, q, \mathcal{F})$ **then**". That is, Algorithm 4 is only called when a subgraph isomorphism is established, which further examines whether $\mathcal{F}$ is valid regarding τ, while the overall algorithm framework remains intact.

Theorem 2 *The partition-based filtering method with enhanced matching condition satisfies the completeness.*

Proof Consider a similar graph pair (g, q) with a given partitioning $P(g)$. By Theorem 1, there exists at least one matching partition from $P(g)$ such that p of g matches q via subgraph isomorphic mapping $\mathcal{F}$; moreover, it requires no more than τ edit operations to transform $g \setminus q$ to $q \setminus \mathcal{F}(p)$. In this case with $\mathcal{F}$ fixed, the number of edit errors when matching the extended partition of p to the (induced) subgraph $\mathcal{F}(p)$ should not be over τ, and hence, it will meet the constraint of enhanced matching condition. In other terms, p is a matching partition under enhanced matching condition. Hence, (g, q) will be identified as a candidate, and thus the completeness of the filtering method is satisfied. □

4.3 Recycling mismatching partitions

While enhanced matching condition exerts a stricter constraint on matching partitions, looking from a different perspective this subsection exploits mismatching partitions.

We call $|\mathcal{F}|$, the cardinality of a (partial) mapping from p to q, the *depth* of the mapping $\mathcal{F}$. Among all the mappings explored by the algorithm, there is a *mismatching depth* d, which equal the minimum of all $|\mathcal{F}|$'s. A full mapping is found if and only if d equals $|V_p|$. Contrarily, if no full mapping is found, it identifies the vertices that make p not contained by q, which are not included in the partial mapping yielding d. In other words, we could have allocated fewer vertices to p, and save this extra portion for other partitions, in which way we hope to make the full use of mismatching partitions. In the sequel, we show how to recycle these vertices and start with an example.

Example 9 Further to Example 6 and Example 7, we first conduct subgraph isomorphism test for p'_1 against q, and no mapping is found for the first vertex P. Thus, $d = 0$ for p'_1. Next, we test p'_2 against q, and a full mapping is found from the original p'_2 to q. In this case, if we recycled the unexplored vertices S, C_1, C_2 as well as incident edges, we could append them to p'_2, and hence, the QISequence of p'_2 becomes as in the right of Fig. 7. The new p'_2 is not contained by q, and consequently, g_2 is no longer a candidate.

The principle of dynamic partition filtering is to leverage the mismatching partition and to dynamically add "unused" vertices and edges to a matching partition. Intuitively, the less we use for the current partition, the more we save for the remaining partitions. To save the maximum for the remaining partitions, we formulate the following problem.

Problem 3 *(minimum mismatching depth) Given a partition p and query graph q, find the minimum mismatching depth $d_{\min}$ such that $p_{d_{\min}} \not\sqsubseteq q$, where $p_{d_{\min}}$ is a vertex-induced subgraph of p induced by the first $d_{\min}$ vertices of V_p as per a partial order $\mathcal{O}$, $d_{\min} \in [0, |V_p| - 1]$.*

Theorem 3 *Problem 3 is NP-hard.*

Proof We consider the decision version of Problem 3, which is stated as follows: Given a partition p and query graph q, is there a mapping $\mathcal{F}$ with mismatching depth no larger than d. A special case of the decision problem is when $d = |\mathcal{F}| - 1$, which is simply the subgraph isomorphism test that is proved to be NP-complete. Therefore, the decision problem is at least as hard as subgraph isomorphism test, and therefore the theorem follows. □

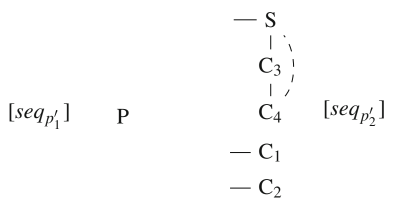


Fig. 7 Example of recycling (solid lines—spanning and half-edges; dashed lines—backward edges)

Having seen the NP-hardness, we present a heuristic method to avoid the high complexity at online query processing. Algorithm 5 implements the subgraph isomorphism test equipped with mismatching partition recycle. d is used to log the mismatching depth, which is initialized to 0 in the first call. If the algorithm returns **false** in the outmost call, d advises that the subgraph induced by the first $d + 1$ vertices is enough to prevent this partition from matching. As a by-product of the subgraph isomorphism test for future use, for every data graph g having p as its partition, we respectively recycle the vertices $v_i \in seq_p, i > d + 1$, as well as their incident edges.

The recycled vertices and edges are utilized once the subgraph isomorphism test invoked by Line 3 of Algorithm 2 returns **true**. In particular, for each data graph g in p's postings list, we append g's recycled vertices and edges to p and perform another subgraph isomorphism test. Only if the new partition is contained by q does g become a candidate to be verified by GED computation. Note that if the new subgraph isomorphism test fails, the vertices and edges beyond $d + 1$ can be recycled again.

Complexity analysis It can be verified that Algorithm 5 correctly computes the containment between p and q, as well as the mismatching depth d. In addition to the isomorphism test, constant time is required to collect the unused subgraph of p by copying and following the QISequence.

Theorem 4 *The partition-based filtering method with mismatching partition recycle satisfies the completeness.*

Algorithm 5: RecyclingSubgraphIso $(p, q, \mathcal{F})$

```
Input  : p is a partition; q is a query graph; F is a mapping
         vector.
Output : A boolean indicating whether $p \sqsubseteq q$.
if $d < |\mathcal{F}|$ then $d \leftarrow |\mathcal{F}|$ ;
if $|\mathcal{F}| = |V_p|$ then return true ;
$v \leftarrow$ next vertex in $seq_p$;
$U \leftarrow \{u \mid u \in$ FindCandidate$(v, seq_p, q, \mathcal{F})\}$;
foreach $u \in U$ do
    $\mathcal{F}' \leftarrow \mathcal{F} \cup \{v \rightarrow u\}$;
    if RecyclingSubgraphIso $(p, q, \mathcal{F}')$ then return true ;
if this is the outmost call then
    foreach $g$ in $I_p$ such that $\mathcal{M}[g]$ is not initialized do
        foreach $v_i \in seq_p, i > d + 1$ do
            add $v_i$ and its incident edges in $g$ into $\Delta_g$;
return false;
```

Proof Without loss of generality, we assume that given a partitioning of a graph g $P = \{p_1, p_2, \ldots, p_{\tau+1}\}$, the dynamic partitioning adjusts p_1 and p_2 to p'_1 and p'_2, respectively. That is, only partial p_1 is used for filtering, and $p_1 \setminus p'_1$ is recycled, which makes p_2 into p'_2, while the remaining partitions are not affected. It can be seen that $P' = \{p'_1, p'_2, \ldots, p_{\tau+1}\}$ is also a partitioning of g. Then, based on Lemma 1, P' can be also used for partition-based filtering without incurring false negatives. Consequently, the repartitioning does not affect the completeness.

In the general case where a succession of adjustments is involved, by using the aforementioned procedure for induction, the final partitioning does not impact on the completeness either. Therefore, the theorem is proved. □

4.4 Comparing filtering methods

We remark to compare the proposed filtering methods in this section with the basic partition-based filtering scheme. In essence, the basic partition-based filtering scheme takes advantage of a given partitioning of data graph. Nonetheless, the two advanced filtering methods share one common point—*adjust* the partitioning of a data graph. Enhanced matching condition exploits the neighborhood proximity of a partition to improve its selectivity. However, such modification is only *logically*, which does not affect the remaining partitions *actually*. On the contrary, mismatching partition recycle saves unused vertices and edges, and dispatches them on demand in future. Thus, it does make a new partitioning by moving the recycled elements to other partitions.

Ascribable to the intrinsic disparity, the filtering constraints of the two methods vary—enhanced matching condition allows the extended partitions to match within a relaxation of τ, while mismatching partition recycle requires the extended partition have an exact match. This is rational, because the *latter* strictly enforces the principle of partition-based filtering that each vertex and edge is in only one partition. In the *former* case, however, the expanded vertices and edges are tested at least twice—respectively, in Δp for enhanced matching condition checking and in p' for basic partition-based filtering. It is hence beyond the applicable scope of the partition-based filtering principle. Therefore, to ensure enhanced matching condition does not admit false negatives, it has to relax the constraint rather than require an exact match of extended partitions.

5 Verification

In this section, we first review a state-of-the-art GED verification algorithm, followed by speedup over it.

5.1 GED Verification

The most widely used algorithm to compute GED is based on A* [17], which explores all possible vertex mappings between graphs in a *best-first* search fashion. It maintains a priority queue of states, each representing a partial vertex mapping $\mathcal{F}$ of the graphs associated with a priority via a function $f(\mathcal{F})$. $f(\mathcal{F})$ is the sum of: (1) $g(\mathcal{F})$, the distance between the partial graphs regarding the current mapping; and (2) $h(\mathcal{F})$, the distance estimated from the current to the goal—a state with all the vertices mapped. In unweighted graphs, $h(\mathcal{F})$ equals the numbers of vertex and edge relabeling between the remaining parts of g and q.

We encapsulate the details in Algorithm 6. It takes as input a data graph, a query graph and a distance threshold, and returns the edit distance if $GED(g, q) \leq \tau$, or $\tau + 1$ otherwise. First, it arranges the vertices of g in an order $\mathcal{O}$ (Line 1), e.g., ascending order of vertex identifers [17]. The mapping $\mathcal{F}$ is initialized empty and inserted in a priority queue $\mathcal{Q}$ (Line 2). Next, it goes through an iterative mapping extension procedure till (1) all vertices of g are mapped with an edit distance no more than τ (Line 6); or (2) the queue is empty, meaning the edit distance exceeds τ (Line 13). In each iteration, it retrieves the mapping with the minimum $f(\mathcal{F})$ in the queue (Line 5). Then, it tries to map the next unmapped vertex of g as per $\mathcal{O}$ (Line 7), to either an unmapped vertex of q, or a dummy vertex to indicate a vertex deletion. Thereupon, a new mapping state is composed, and evaluated by ExistingDistance and EstimateDistance to calculate the values of $g(\mathcal{F})$ and $h(\mathcal{F})$, respectively. ExistingDistance carefully checks how many edit error there are between the subgraph of g induced by vertices already in $\mathcal{F}$ and its counterpart of q. EstimateDistance gives a lower bound of potential edit errors that will be incurred by matching the remaining graphs. $g(\mathcal{F})$ and $h(\mathcal{F})$ together makes $f(\mathcal{F})$, an estimation

Algorithm 6: GEDVerification (g, q)

```
   Input  : $g$ is a data graph; $q$ is a query graph.
   Output : $GED(g, q)$, if $GED(g, q) \leq \tau$; or $\tau + 1$, otherwise.
1  $\mathcal{O} \leftarrow$ order the vertices of $g$;
2  $\mathcal{F} \leftarrow \emptyset$, $\mathcal{Q} \leftarrow \emptyset$;
3  $\mathcal{Q}.push(\mathcal{F})$;
4  while $\mathcal{Q} \neq \emptyset$ do
5      $\mathcal{F} \leftarrow \mathcal{Q}.pop()$;
6      if $|\mathcal{F}| = |V_g|$ then return $g(\mathcal{F})$ ;
7      $u \leftarrow$ next unmapped vertex in $V_g$ as per $\mathcal{O}$;
8      foreach $v \in V_q$ such that $v \notin \mathcal{F}$ and
       $|deg(u) - deg(v)| \leq \tau$ or a dummy vertex do
9          $\mathcal{F} \leftarrow \mathcal{F} \cup \{u \rightarrow v\}$;
10         $g(\mathcal{F}) \leftarrow$ ExistingDistance($\mathcal{F}$);
11         $h(\mathcal{F}) \leftarrow$ EstimateDistance($\mathcal{F}$);
12         if $f(\mathcal{F}) = g(\mathcal{F}) + h(\mathcal{F}) \leq \tau$ then $\mathcal{Q}.push(\mathcal{F})$;
13 return $\tau + 1$;
```

of overall edit distance that never exceeds the real value, and only if $f(\mathcal{F}) \leq \tau$ is the state inserted into the queue (Lines 9–12).

Definition 7 (*Correctness*) Given a candidate pair (g, q), a verification algorithm is correct, provided it satisfies (1) if (g, q) passes the algorithm, (g, q) must be a similar pair, and (2) if (g, q) is a similar pair, it must pass the algorithm.

While it is straightforward to verify that Algorithm 6 satisfies the correctness, the search space of the algorithm is exponential in the number of vertices. More importantly, it overlooks the option of leveraging the matching partitions obtained from candidate generation. In other words, there are opportunities to reuse the computation for matching partitions to speed up the verification. Next, we present our improvement by sharing computation between dynamic partition filtering and candidate verification.

5.2 Extending matching partition

Recall Algorithm 2 admits a list of graphs as candidates if the corresponding partition of the postings list is contained by the query via subgraph isomorphism test. As each g in the list shares with q a common subgraph, i.e., the matching partition, we can use this common part as the starting point to verify the pair. Based on this intuition, we devise a verification algorithm by extending the matching partitions.

The basic idea of the extension-based verification technique is to fix the existing mapping $\mathcal{F}$ between the matching partition p and q from the subgraph isomorphism test in the filtering phase, and further match the remaining subgraph $g \setminus p$ with $q \setminus \mathcal{F}(p)$ using Algorithm 6 in the verification phase. In order not to miss real results, if g has multiple matching partitions, we need to run such procedure multiple times, each starting with a mapping from p to q, where p is a matching partition surviving the filters.

Example 10 Consider the data graph g with its two partitions and a query graph q shown in Fig. 8, and $\tau = 1$. The partition -C-O$_1$ is contained by q via a mapping to either -C-O$_1$ or -C-O$_2$. To carry out the extension-based verification, assume the first mapping is to -C-O$_1$, and then we try to match N and O$_2$ in succession. After it fails to find a mapping with GED within τ, we proceed with the next mapping -C-O$_2$, which is invalid either. Eventually, we can verify g is not an answer since $GED(g, q) \geq 2$.

Algorithm 7 outlines the extension-based verification. It takes as input a data graph g, a query q, a set of matching partitions P from the filtering phase, and the vertex mapping $\mathcal{F}$ obtained via subgraph isomorphism test. Then, it enumerates all possible mappings of p in q, and computes GED starting with the mapping. If a distance in Line 4 is

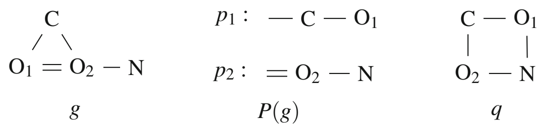


Fig. 8 Example of extension-based verification

no larger than τ, it confirms that there is a valid mapping such that using no more than τ edit operations can transform g to q; otherwise, it proceeds with the next mapping until all mappings are attempted. In each verification, we make minor changes according to Algorithm 8, and let it take as input a mapping $\mathcal{F}$, either retrieved from the filtering phase in Line 2 of Algorithm 7 or enumerated in Line 6 of Algorithm 7. In particular, $g(\mathcal{F})$ and $h(\mathcal{F})$ are computed first, and $\mathcal{F}$ is inserted as the initial state into the priority queue, if $f(\mathcal{F})$ does not exceed the threshold. Hence, the remaining unmapped vertices of g, i.e., $V_g \setminus V_p$, are given an order and processed according the classic A* algorithm. Since the vertices in $V_g \setminus V_p$ remain unchanged for a matching partition p, the order is cached during various mappings of p. After testing all matching partitions $p \in P$, in which all subgraph isomorphic mappings from p to q are tried, it comes to the conclusion of **false** indicating $GED(g, q) > \tau$.

Complexity analysis During verification, we need to establish a mapping for each vertex v of q, where in each step, v finds a match, which can be any vertex of g or null, and edge correspondence also needs to be checked. In all, the worst case complexity is $O((|V_q| \cdot (|V_g| + |E_g| + \gamma_g))^{|V_g|})$, which is identical to that of Algorithm 6.

We remark that the empirical number of search states of our solution is usually much smaller than that of the classic A* algorithm (to be experimentally verified in Sect. 8.3). By fixing the matching partition p to $\mathcal{F}(p)$, we only match

Algorithm 7: ExtensionBasedDist (g, q, P)

```
1 foreach matching partition $p \in P$ do
2     $\mathcal{F} \leftarrow$ mapping of $p$ from filtering phase;
3     while $\mathcal{F} \neq \emptyset$ do
4         distance $\leftarrow$ GEDVerification$(g, q, \mathcal{F})$;
5         if distance $\leq \tau$ then return true ;
6         else $\mathcal{F} \leftarrow$ EnumerateNextMap$(p, q)$;
7 return false;
```

Algorithm 8: Replacement of Lines 2 and 3 of Algorithm 6

```
1 $g(\mathcal{F}) \leftarrow$ ExistingDistance$(\mathcal{F})$ ;    /* $\mathcal{F}$ is a subgraph
  isomorphic mapping of $p$ in $q$ */
2 $h(\mathcal{F}) \leftarrow$ EstimateDistance$(\mathcal{F})$;
3 if $f(\mathcal{F}) = g(\mathcal{F}) + h(\mathcal{F}) \leq \tau$ then
4     $\mathcal{O} \leftarrow$ order the vertices in $V_g \setminus V_p$ ;    /* cached */
5     $\mathcal{Q}.push(\mathcal{F})$;
```

an unmapped vertex in $g \setminus p$ to a vertex in $q \setminus \mathcal{F}(p)$; if p has more embeddings in q, the cost of locating multiple embeddings is also much smaller via subgraph isomorphism. In our implementation, instead of collecting all the matching partitions in P, we conduct verification (Lines 3–5) of g against q once we find a matching partition that passes the dynamic partition filtering. If it returns **true**, g is an answer and excluded from future computation; otherwise, it resumes the index probing and filtering. By doing this, if we encounter an index entry with empty inverted list, we skip directly to the next one. Therefore, the extension-based solution not only shrinks the search space, but also shares the computation between the filtering and the verification phases.

Theorem 5 *The extension-based verification satisfies the correctness.*

Proof It is can be seen that if g is not an answer with respect to q, there does not exist a mapping between g and q in any case, and thus, the verification procedure returns **false**.

If the extension-based verification returns **true**, it means that there exists at least one mapping $\mathcal{F}$ between g and q such that using no more than τ, say θ, edit operations can transform g to q based on $\mathcal{F}$. It may be or may not be the optimal mapping, but we are assured that $GED(g, q) \leq \theta \leq \tau$. In other words, g is an answer in this case, though $\mathcal{F}$ may not be the one yielding the minimum number of edit operations, or equally, the algorithm may not provide the real GED. □

5.3 Completeness and correctness

All the proposed techniques constitute our advanced Pars algorithm for threshold-based similarity search. We show the completeness and correctness of the algorithm.

Theorem 6 *The proposed algorithm Pars satisfies*

- *Completeness: given any similar pair (g, q), our algorithm must find it as an answer;*
- *Correctness: a pair (g, q) found by our algorithm must be a similar pair.*

Proof We first prove completeness. Based on Theorems 1, 2 and 4, given a similar pair (g, q), the basic partition-based filtering and dynamic partition filtering schemes must find this pair as a candidate pair. Based on Theorem 5, (g, q) can pass our extension-based verification. Thus, it must be identified and added to the answer set, and hence, the algorithm satisfies the completeness.

Next, we show correctness. Guaranteed by Theorem 5, any pair having passed our extension-based verification must be a similar pair conforming the distance constraint. Therefore, our algorithm satisfies the correctness. □

6 Cost-aware graph partition

In this section, we propose a cost model to analyze the effect of graph partitioning on query processing, based on which a practical partitioning algorithm is devised.

6.1 Effect of graph partitioning

Recall Algorithm 2. It tests subgraph isomorphism from each indexed partition p to the query graph q. Ignoring the effect of size filtering, label filtering and dynamic partition filtering, graphs in the postings list of p are included as candidates, if $p \sqsubseteq q$. Therefore, the candidate set $C_q = \cup_p \{ D_p \mid p \sqsubseteq q,\ p \in \mathcal{P} \}$, where $D_p = \{ g \mid p \sqsubseteq g,\ g \in G \}$. Incorporating a binary integer φ_p to indicate whether $p \sqsubseteq q$, we rewrite the candidate number as

$$|C_q| = \sum_p \varphi_p \cdot |I_p|,\ p \in \mathcal{P},$$

where I_p is the postings list of p. Suppose there is a query workload Q, and denote ϕ_p as the probability that $p \sqsubseteq q,\ q \in Q$; i.e., $\phi_p = \frac{|\{ q | p \sqsubseteq q \wedge q \in Q \}|}{|Q|}$. The expected number of candidates of a query $q \in Q$ is

$$|\overline{C_q}| = \sum_p \phi_p \cdot |I_p|,\ p \in \mathcal{P}.$$

Since the postings lists are composed of data graph identifiers, we rewrite it using a binary integer variable π_g^p,

$$|\overline{C_q}| = \sum_g \sum_p \phi_p \cdot \pi_g^p,\ p \in \mathcal{P},\ g \in G,$$

where π_g^p is 1 if p is one of g's partitions, and 0 otherwise.

We interpret the expected candidate number as a commodity contributed by all data graphs. As g is partitioned into $\tau + 1$ partitions $P = \{ p_i \}, i \in [1,\ \tau + 1]$, the expected number of contributed candidates from a data graph g is

$$\overline{c_g} \triangleq c_P = \sum_{i=1}^{\tau+1} \phi_{p_i} \cdot |G|. \tag{1}$$

In light of this, we observe that data graphs are mutually independent for minimizing candidates from a partition-based index. Immediate is that

$$|\overline{C_q}| = \sum_g \overline{c_g},\ g \in G.$$

Example 11 Consider in Fig. 9 data graph g to be partitioned, the three graphs below as $Q = \{ g_1, g_2, g_3 \}$, and $\tau = 1$. A partitioning $P(g)$ is presented besides g. Testing p_1 against

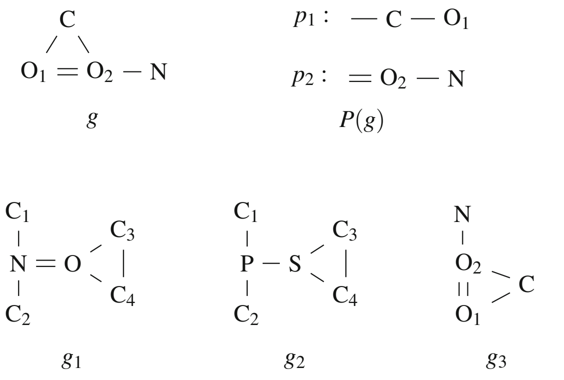


Fig. 9 Effect of different partitionings

Q confirms that no graph in Q contains p_1, and thus $\phi_{p_1} = 0$; similarly, $\phi_{p_2} = 0$. $c_P = (\phi_{p_1} + \phi_{p_2}) \cdot |G| = 0$. Moving vertex O_1 from p_1 to p_2 yields $P' = \{ p'_1, p'_2 \}$. $c_{P'} = (\phi_{p'_1} + \phi_{p'_2}) \cdot |G| = (3/3 + 0) \cdot |G| = |G|$. P is better than P' in terms of Equation (1). In fact, it can be verified that P is one of the best partitionings of g regarding Q.

In case that a historical query workload is not available, we may, as an alternative, sample a portion of the database to act as a surrogate of Q. To this end, a sample ratio ρ is introduced to control the sample size $|Q| = \rho \cdot |G|$. We extract graphs from the database as queries in our experimental evaluation. Thus, we adopt this option so that the index is built to work well with these queries. We also investigate how ρ influences the performance (Sect. 8.5).

Now, we are able to minimize the total number of candidates by minimizing the candidate number from each data graph. We will show our solution in the sequel.

6.2 A practical partitioning algorithm

We formulate the graph partitioning of index construction as an optimization problem.

Problem 4 *(minimum graph partitioning) Given a data graph g and a distance threshold τ, partition the graph into $\tau + 1$ subgraphs such that Equation (1) is minimized.*

As expected, even for a trivial cost function, e.g., the average number of vertices of the partitions, the above optimization problem is NP-hard.[4] Seeing the difficulty of the problem, we propose a practical algorithm as a remedy to select a good partitioning: first randomly generate a partitioning of the data graph and then refine it.

[4] The special case of $\tau = 1$ is polynomially reducible from the *partition problem* that decides whether a given multiset of numbers can be partitioned into two subsets such that the sums of elements in both subsets are equal, and thus, is NP-hard.

Algorithm 9: RandomPartition (g, τ)

```
Input  : $g$ is a data graph; $\tau$ is an edit distance threshold.
Output : A graph partitioning $P$, initialized as $\emptyset$.
1  $M \leftarrow$ empty map from vertex identifier to boolean ;
   /* whether a vertex has been considered */
2  for $i \in [1, \tau + 1]$ do
3      randomly choose a vertex $v \in V_g$ such that $M[v] =$ false;
4      $p_i \leftarrow (\{ v \}, \emptyset, \{ l_v \})$;
5      $M[v] \leftarrow$ true;
6  while $\exists v$ such that $M[v] =$ false do
7      foreach $p_i \in P$ do
8          $u \leftarrow$ ChooseVertexExpand $(p_i)$;
9          ExpandInducedSub $(p_i, u)$;
10 while $\exists (u, v)$ with end vertices in different partitions do
11     randomly assign $(u, v)$ to either $p_u$ or $p_v$;
12 return $P$;
```

Algorithm 10: RefinePartition (P, Q)

```
Input  : $P$ is a graph partitioning; $Q$ is a set of query graphs.
Output : $P$ is an optimized graph partitioning.
1  updated $\leftarrow$ true;
2  $c_g \leftarrow$ ComputeSupport $(P, Q)$;
3  while updated = true do
4      $c_{\min} \leftarrow c_g$;
5      foreach $(u, v) \in E_g$ do
6          $P' \leftarrow P$;
7          $p'_u \leftarrow$ ShrinkInducedSub$(p'_u, u)$;
8          $p'_v \leftarrow$ ExpandInducedSub$(p'_v, u)$;
9          randomly assign remaining edges between $p'_u$ and $p'_v$;
10         $c'_g \leftarrow$ ComputeSupport$(P', Q)$;
11         if $c'_g < c_{\min}$ then $P_{\min} \leftarrow P'$, $c_{\min} \leftarrow c'_g$ ;
12     if $c_{\min} < c_g$ then $P \leftarrow P_{\min}$, $c_g \leftarrow c_{\min}$;
13     else updated $\leftarrow$ false;
14 return $P$;
```

Algorithm 9 sketches the random partitioning phase, which takes a data graph and a distance threshold as input, and produces $\tau + 1$ partitions as per Definition 4. It maintains a Boolean map M to indicate the vertex states—**true** if a vertex has been assigned to a subgraph, and **false** otherwise. Firstly, it randomly distributes $\tau + 1$ distinct vertices into p_i, $i \in [1, \tau + 1]$ (Lines 2–5). This ensures every p_i is non-empty and contains at least one vertex. Then, for each p_i, we extend it by 1-hop via ChooseVertexExpand, which randomly selects a vertex $v \in V_{p_i}$, chooses to include in p_i an extension vertex u via edge (v, u) that has not been assigned to any partitions, together with its edges connected to the vertices that are already in p_i. If v fails to extend p_i, we select one of v's neighbors in p_i to replace v, and try the expansion again till there is no option to grow (Lines 6–9). This offers each p_i a chance to grow, and hence the sizes and the selectivities of the partitions are balanced. Finally, it assigns the remaining edges (u, v), whose end vertices are in different partitions, randomly to either p_u or p_v as half-edges.

In the refine phase, we take the opportunity to improve the quality of the initial partition, as shown in Algorithm 10. It takes as input a graph partitioning P and a workload of query graphs Q, and outputs the optimized partitioning. Our algorithm optimizes the current partitioning by selecting the best option of moving a vertex u from one partition p_u to another p_v such that $(u, v) \in E_g$. In particular, Line 7 removes u from p'_u by excluding u and its incident edges in p'_u, where p'_u is the partition containing u. Then, in Line 8, it adds u and edges between u and vertices in p'_v. Afterward, the remaining extracted edges are randomly assigned to either p'_u or p'_v as half-edges, since they have end vertices in both partitions. Hence, we have a new partitioning P'. c'_g is computed in Line 10. If it is less than the current best option $c_{\min}$, we replace $c_{\min}$ with c'_g. As a consequence, the best option that reduces c_g the most is taken as the move for the current iteration in Line 12. The above procedure repeats until c_g cannot be improved by $c_{\min}$. To evaluate c_g and c'_g in Lines 2 and 10, respectively, we can conduct subgraph isomorphism tests to collect partitions' support in Q, fulfilled by ComputeSupport.

Correctness and complexity analysis Algorithms 9 and 10 constitute the implementation of GraphPartition. Immediate is that GraphPartition computes a graph partitioning conforming to Definition 4. For Algorithm 9, it takes $O(V + E)$ time to assign vertices and edges. The complexity of Algorithm 10 is mostly determined by ComputeSupport, which carries out subgraph isomorphism tests from the partitions to Q. In each iteration of refinement, we need to conduct $|E|$ rounds of ComputeSupport, through which the supports of two newly constructed partitions are evaluated.

7 Supporting Top-*k* search

This section expounds how to extend the partition-based filtering scheme to handle top-k graph similarity search.

A straightforward way to process top-k queries via threshold-based similarity search is to issue a series of similarity queries, starting at $\tau = 0$ with increment as 1, till there are k graphs in the answer set. However, this naïve solution can be fairly computationally expensive. Next, we devise a hierarchical inverted index by adapting the partition-based inverted index, and then present a novel top-k procedure.

7.1 Agglomerative index organization

We first describe the adaptation of the partition-based inverted index. The partitioning of a graph with $\theta + 1$ partitions is called a *θ-partitioning* of the graph. The basic idea is constituted of two steps: (1) for each data graph, we construct a hierarchy with a $\tau_{\max}$-partitioning as the leaves (level-$\tau_{\max}$), and the graph itself as the root (level-0); and (2) level-wise merge and organize the partitionings of all data graphs.

Specifically, each data graph is associated with a $(\tau_{\max} + 1)$-level hierarchy (level-0 to level-$\tau_{\max}$) such that

- for level-0, it is a dummy node representing g;
- for level-i, it is an i-partitioning of g; and
- one and only one of the partitions at level-i is made of two partitions at level-$(i + 1)$, $i \in \{1, \ldots, \tau_{\max} - 1\}$.

```
Algorithm 11: HierarchicalIndex ($G$, $\tau_{\max}$, $Q$)
  Input  : $G$ is a set of data graph; $\tau_{\max}$ is the maximum edit
           distance threshold; $Q$ is a set of query graphs.
  Output : A hierarchical inverted index $H$.
1  $H \leftarrow \emptyset$;
2  foreach data graph $g \in G$ do
3      $P^{\tau_{\max}} \leftarrow$ GraphPartition($g$, $\tau_{\max}$, $Q$);
4      $i \leftarrow \tau_{\max} - 1$, $h_g \leftarrow P^{\tau_{\max}}$;
5      while $i > 0$ do
6          $c^i \leftarrow \infty$, $P^i \leftarrow \emptyset$;
7          while $P' \leftarrow$ EnumerateNextPart($P^{i+1}$) do
8              $c' \leftarrow$ ComputeSupport($P'$, $Q$);
9              if $c' < c^i$ then $P^i \leftarrow P'$, $c^i \leftarrow c'$;
10         $h_g \leftarrow h_g \cup P^i$;
11     level-wise merge $h_g$ with $H$;
12 return $H$;
```

Then, we adopt an *agglomerative* organization of the partitionings of single graphs. In essence, the first step is comparative to drawing a dendrogram over the $\tau_{\max}$-partitioning of a graph, and the second step is a typical merging process akin to agglomerative clustering [9]. Recall that agglomerative clustering every time merges two elements in the set into one cluster. Hence, we devise an agglomerative method to construct the index as Algorithm 11.

Algorithm 11 takes as input a set of data graphs G, the maximum threshold $\tau_{\max}$ to support, and a query workload Q, and produces a hierarchical index H. It processes all the data graphs one-by-one. For each g, taking the $\tau_{\max}$-partitioning as the base (Line 3), we add it to the hierarchy first, and then, construct the upper levels by iterations (Lines 5–10). The core procedure resembles the refinement of Algorithm 10 that, at level-i, it composes a new partitioning by combining two partitions from level-$(i+1)$. By testing all possible combinations of the partitionings at level-$(i + 1)$, the best one is retained as the partitioning at level-i. This heuristic ensures that level-i is always an i-partitioning of g, and hence, the partition-based filtering scheme can be appropriately applied. Then, the level-i index is added to be hierarchy h_g, and another level-up round for level-$(i - 1)$ begins thereafter until it reaches the root of the hierarchy, which completes the index construction of a single graph.

After constructing the hierarchies of single data graphs, we merge them level-by-level (Line 11). We denote the partitioning of graph g at level-θ by h_g^θ. In particular, two hierarchies h_g and $h_{g'}$ are processed as follows. Starting from level-$\tau_{\max}$, we make a *set* of subgraphs, each of which is from the partitions at level-$\tau_{\max}$ of h_g and $h_{g'}$. Specifically for level-i, $i \in \{1, \ldots, \tau_{\max}\}$, if a partition from h_g^i collides with another from $h_{g'}^i$, they correspond to the same subgraph at level-i of the index (as entries of dictionary in inverted index), and g and g' are inserted into a list attached to that subgraph (as postings list in inverted index); otherwise, a new dictionary entry is created for that partition, and g is inserted into a list of the entry. After merging all hierarchies of single data graphs, all the data graphs and their partitions are organized by a hierarchical inverted index. Note that the linage of the original hierarchies are retained in the index. This completes the index construction, and next, we look into the search procedure.

7.2 Top-k search procedure

Inspired by classic top-k search algorithms, we maintain a priority queue $\mathcal{Q}$ to keep the current best k results. Let $U_\mathcal{Q}$ denote the largest distance between q and the k graphs in $\mathcal{Q}$. It is observed that the distance between any graph in the top-k results to the query graph cannot exceed the current $U_\mathcal{Q}$ when $\mathcal{Q}$ is filled up. Thus, we can prune a data graph if its distance to the query graph is determined to be larger than or equal to $U_\mathcal{Q}$. Nevertheless, in contrast to the straightforward solution via multiple threshold-based similarity queries, we propose to first access the promising graphs that have large possibility to be similar to the query graph, e.g., graphs sharing many common partitions with the query graph. Based on the promising graphs, we can accurately estimate an upper bound of distances of top-k answers to the query graph, which can then be used for pruning dissimilar graphs.

To illustrate how to leverage this pruning idea, we present Algorithm 12, which takes as input a query graph q, an integer k as well as a hierarchical inverted index H, and reports a list $\mathcal{Q}$ of k graphs with the smallest distance to q.

It first initializes a priority queue $\mathcal{Q}$ for keeping the current best k graphs (Line 1). Then, it probes the hierarchical inverted index level-by-level, from 1 to $\tau_{\max}$. Inside the iterations, there is an exit condition that $U_\mathcal{Q}$ is no larger than the maximum threshold supported by level-i (Line 3). Recall that level-i index supports threshold-based similarity search with threshold up to i; in other words, all graphs with distance less than $i - 1$ have been dealt with in previous rounds, and the current round is for graphs with distance greater than or equal to i. If $U_\mathcal{Q} \geq i$, it essentially implies all unseen data graphs are of distance at least $U_\mathcal{Q}$. In this case, top-k results shall be found by now; otherwise, we continue to probe H^i to find graphs that have at least one matching partition. By the principle of partition-based filtering, only these graphs can make candidates to the top-k results. Moreover, these graphs are grouped into a number of sets according to the overall size of matching partitions (Line 4). Each set is denoted by N_j, and j is the total number of *vertices and edges* of matching partitions. Subsequently, for every graph of the sets with j from large to small, we first test whether the graphs can pass the *size filter* (Line 6). That is, if $|V_q| + |E_q| - j \geq U_\mathcal{Q}$, the graphs in N_j cannot match q with edit distance smaller than $U_\mathcal{Q}$. In addition, we can prune the graphs in N_k with $k < j$ as well, by skipping the remaining sets and proceeding to the next iteration. Then, we verify the distance from each graph in N_j to the query graph, starting from the one with most *matching partitions*, and update $\mathcal{Q}$ and $U_\mathcal{Q}$ if necessary (Lines 5–9). The intuition is that the larger j, the more common elements between q and $g \in N_j$; the more matching partitions, the larger chance for g in top-k results. Iterating in this way, we expect to find the right graphs earlier.

Algorithm 12: TopkSearch (q, k, H)

```
Input  : $q$ is a query graph; $k$ is an integer; $H$ is a hierarchical
         inverted index.
Output : $\mathcal{Q}$ is a list of $k$ graphs with smallest distance to $q$.
1  $\mathcal{Q} \leftarrow \emptyset, U_\mathcal{Q} \leftarrow \infty$;
2  for $i = 1 \rightarrow \tau_{\max}$ do
3      if $i \geq U_\mathcal{Q}$ then return $\mathcal{Q}$;
4      probe $H^i$ to find graphs with at least one matching partition,
       and group them into $N_j$'s;
5      foreach $N_j$ with $j$ from large to small do
6          if $|V_q| + |E_q| - j \geq U_\mathcal{Q}$ then break ;
7          foreach graph $g \in N_j$ with number of matching
           partitions from large to small do
8              if GEDVerification$(g, q, U_\mathcal{Q}) < U_\mathcal{Q}$ then
9                  update $\mathcal{Q}$ and $U_\mathcal{Q}$;
10 return $\mathcal{Q}$;
```

The advantage of the proposed search procedure mainly comes from the fact that it (1) filters out graphs in a batch mode by grouping as per matching size, and (2) derives a tight upper bound $U_\mathcal{Q}$ quickly for pruning and verification.

Look-ahead While the aforementioned pruning is effective when each mismatching partition contains only one edit error, this estimation can be inaccurate when there are more edit errors inside single mismatching partitions. Inspired by the filtering principle of Corollary 1, we conceive a look-ahead strategy to prune by leveraging the hierarchical index.

Example 12 Consider in Fig. 10 data graph g and query graph q, and assume we are probing level-2 of the index, and the current $U_\mathcal{Q}$ is 4. The hierarchy of g is shown on the left, where "●" means that it is a partition identical to the one below at the lower level. We observe that all 3 partitions—p_1, p_2 and p_3—do not match q, giving a distance lower

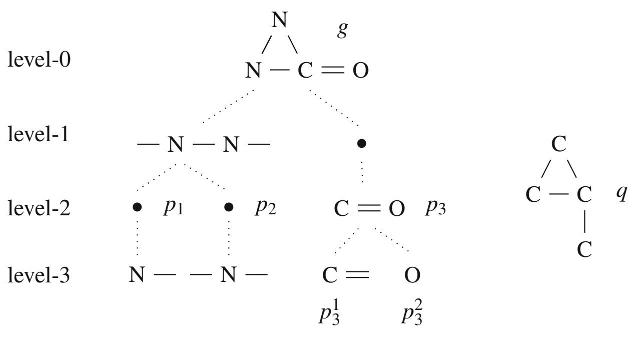


Fig. 10 Example of look-ahead strategy

bound 3, less than U_Q. Hence, g needs to be verified in group N_0. However, if we look one level down to level-3, we find that C=O is broken into p_3^1 and p_3^2, which neither match q. Thus, they, together with p_1 and p_2, form a 4-partitioning of g and produce a distance lower bound 4, equal to U_Q. Thus, we disqualify g from results without verification.

The example manifests the chance of filtering out more non-promising candidates by looking into the subsequent levels under mismatching partitions. In general, consider a data graph g at level-i with at least one matching partition, $i \in \{1, \dots, \tau_{\max}\}$, and we cannot exclude g. Instead of immediately verifying g against q, we further divide the mismatching partitions following the hierarchy of g, and estimate a potentially tighter distance lower bound by testing the new partitions. The intuition is that the cluttered edit errors may be separated into smaller subgraphs at lower levels, and hence, identified by partition-based filtering. In other words, if the number of mismatching partitions is increased to be greater than or equal to U_Q with the new partitioning, we are able to prune g (though it might be discovered again later when i increases and the top-k results are not filled up). Thus, this pruning effectively postpones verification, until we cannot look ahead further, i.e., to the lowest level, in which case we execute final verification. Note that we only examine and look ahead for mismatching partitions in the subsequent levels, since matching partitions never contribute edit errors later on.

```
Algorithm 13: LookAhead ($g, q, U_Q, i$)
  Input  : $g$ is a data graph; $q$ is a query graph; $U_Q$ is a distance
           threshold; $i$ is the current index level.
  Output : A boolean - false if $GED(g, q) \geq U_Q$; true
           otherwise.
1 while $i \leq \tau_{\max}$ do
2     examine partitions whose parents mismatch $q$;
3     if more mismatching partitions are found then
4         $lb \leftarrow$ number of mismatching partitions at level-$i$;
5         if $lb \geq U_Q$ then return false ;
6     $i \leftarrow i + 1$;
7 return true;
```

We call this *look-ahead* strategy, abstracted in Algorithm 13, and apply it right before verification. In our implementation, we employ a queue to keep the mismatching partitions of g. Every time, we take the first partition in the queue, and find the two subgraphs composing it according to the hierarchical index. Then, we test each of them against q, and if it mismatches, we increase by 1 the lower bound. If the lower bound is no less than U_Q, we can safely prune g and exit the iterations; otherwise, we insert it into the end of the queue, and start another round with the first element in the queue.

Computation sharing As a further optimization, we propose to reuse the computation during look-ahead for subsequent levels. Recall that a data graph g pruned at level-i may be discovered again at level-j, $i < j \leq \tau_{\max}$. To this end, when conducting the look-ahead strategy, we record the distance lower bound lb for g, and g enters look-ahead again or final verification only if i becomes greater than lb. This saves g from potential repeated look-ahead pruning. I,n addition, as some of the partitions are verified during the look-ahead for g, we make a note of the matching results on the corresponding entries in the hierarchical inverted index. By doing this, afterward when we carry out index probing on following levels as well as look-ahead, the marked partitions are not to be tested again, and the matching information is hence reused, avoiding computational redundancy.

For final verification, we call the basic A*-based algorithm using U_Q as threshold. As a minor optimization, when U_Q is too large or $\mathcal{Q}$ is not yet filled up, we employ an existing upper-bounding technique [6,29] to restrict the search space, since the real distance never exceeds the upper bound.

8 Experiments

This section reports experimental results and our analyses.

8.1 Experiment setup

We conducted experiments on 3 public real-life datasets:

- **AIDS** is an antivirus screen compound dataset from the Developmental Therapeutics Program at NCI/NIH.[5] It contains 42,687 chemical compound structures.
- **PROTEIN** is a protein database from the Protein Data Bank,[6] constituted of 600 protein structures. Vertices rep-

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

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

Table 2 Dataset statistics

Dataset	$\lvert G\rvert$	avg $\lvert V\rvert/\lvert E\rvert$	$\lvert l_V\rvert/\lvert l_E\rvert$	γ
AIDS	42,687	25.60/27.60	62/3	12
PROTEIN	600	32.63/62.14	3/5	9
NASA	36,790	33.24/32.24	10/1	245

resent secondary structure elements, labeled by types; edges are labeled with lengths in amino acids.
– **NASA** is an XML dataset storing metadata of an astronomical repository,[7] including 36,790 graphs. We randomly assigned 10 vertex labels to the graphs, as the original graphs are nearly of unique vertex labels.

Table 2 lists the statistics of the datasets. AIDS is a popular benchmark for structure search, PROTEIN is denser and less label-informative, and NASA has more skewed vertex degree distribution. We randomly sampled 100 graphs from every dataset to make up the corresponding query set, which were used throughout the experimental studies for performance evaluation. Thus, the queries are of similar label distribution and average density to the data graphs. The average $|V_q|$ for AIDS, PROTEIN and NASA are 26.70, 31.67 and 42.51, respectively. In addition, the scalability tests involve synthetic data, which were generated by a graph generator.[8] It measures graph size in terms of $|E|$, and density is defined as $d = \frac{2|E|}{|V|(|V|-1)}$, equal 0.3 by default. The cardinalities of vertex and edge label domains are 2 and 1, respectively.

Experiments were conducted on a machine of Quad-Core AMD Opteron Processor 8378@800 MHz with 96G RAM,[9] running Ubuntu 10.04 LTS. All the algorithms were implemented in C++, and ran in main memory. We evaluated threshold-based queries with $\tau = \tau_{\max}$ at both indexing and query processing stages, and top-k queries with $\tau_{\max} = 6$ at indexing. We measured

– index size by memory space consumption;
– index construction time;
– number of candidates that need GED verification; and
– query response time,[10] *including* candidate generation and GED verification.

Candidate number and running time are reported on the basis of the 100 queries.

[7] http://www.cs.washington.edu/research/xmldatasets/.

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

[9] This RAM configuration is to accommodate the A*-based verification algorithm, which needs to maintain a large number of partial mappings in a priority queue.

[10] Some of the experiments were manually terminated after running for 24 h, the duration of which was reported instead.

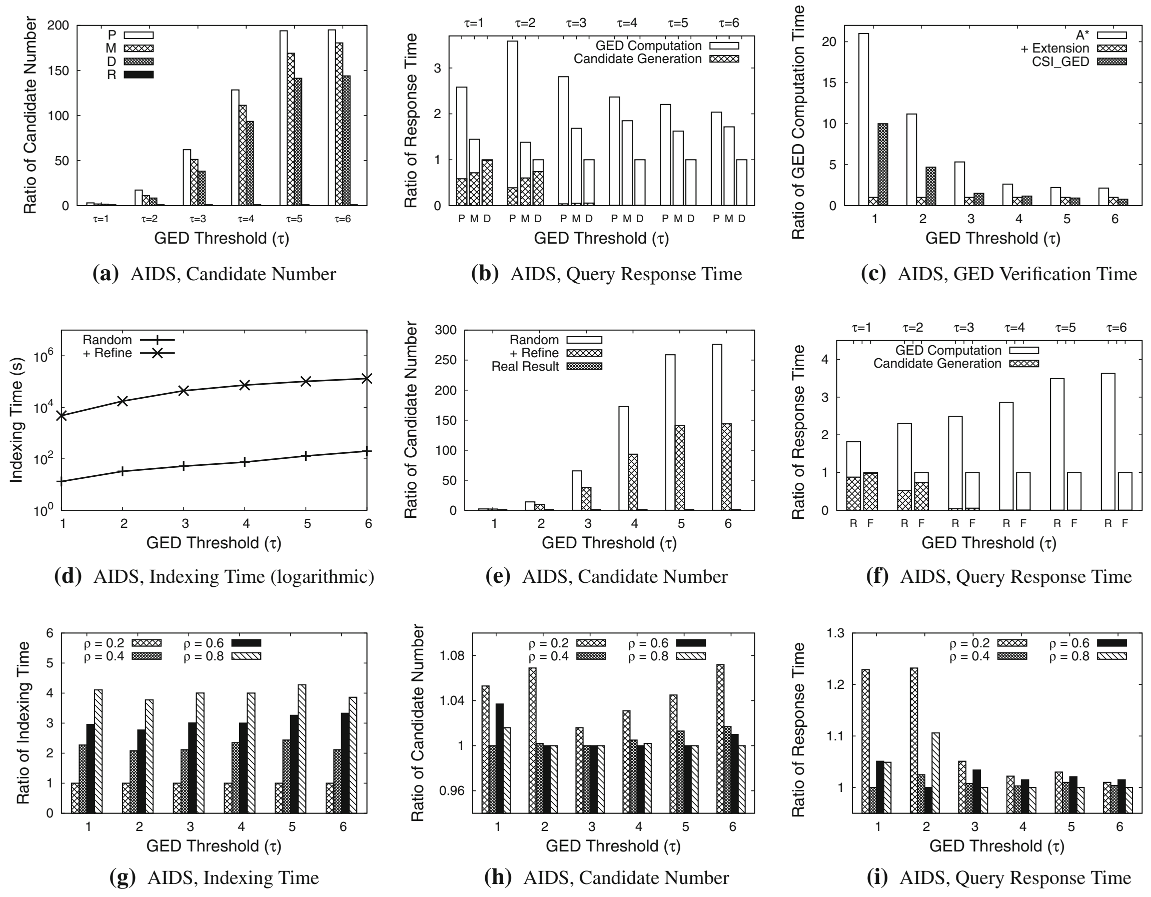
8.2 Evaluating filtering methods

We first evaluate the proposed filtering methods. We use "Basic Partition" to denote the basic implementation of our partition-based similarity search algorithm, "+ Match" to denote the basic algorithm equipped with the enhanced matching condition, and "+ Dynamic" to denote the implementation of integrating + Match with dynamic partition filtering.

Figure 11a summarizes candidate numbers on AIDS, where "P", "M", "D" and "R" are short for Basic Partition, + Match, + Dynamic and real result, respectively. In particular, it compares the ratios of candidate numbers to the real result number; that is, for each GED threshold, the real result number is used as the base (i.e., 1 in the figure), and the quotient of candidate number divided by the base is shown.[11] Specifically, when τ is the largest as 6, there are in total $|G| \times (\tau + 1) = 42,687 \times 7 = 298,809$ partitions, and on average 26,951 out of the 45,263 distinct partitions (59.5%) hit the query by Basic Partition. Moreover, the general trend is that candidates returned by the three methods increase with the growth of τ, and the gap is more remarkable when τ is large. The trend is within expectation according to the discussion in the end of Sect. 4. As read from the results, candidate set shrinks when additional filtering techniques + Match and + Dynamic are applied. The margin is substantial, especially between + Dynamic and Basic Partition; when $\tau = 1$, + Dynamic provides a reduction over + Match and Basic Partition by 32 and 51%, respectively.

To reflect the filtering effect on response time, we appended the basic A* algorithm (denoted "A*") to verify the candidates. The query response time is summarized in Fig. 11b, where the running time of + Dynamic is used as the base for each GED threshold. The filtering time of + Dynamic is greater than + Match, and + Match is slightly greater than Basic Partition; whereas, as an immediate consequence of less candidates, the overall response time of + Dynamic is smaller by up to 67 and 36%, respectively, in comparison with Basic Partition and + Match, among all the thresholds. Thus, enhanced matching condition benefits the overall performance while incurring a little overhead in filtering time; mismatching partition recycle needs more computation in filtering but remarkably improves the overall runtime performance in return. Note that since candidate generation is fairly fast compared with GED verification, its running time becomes almost invisible for $\tau \geq 3$.

[11] Onwards this method is adopted for summarizing experiment results if not otherwise specified, and figures of the same results in logarithmic scale can be found in Part C of supplementary material to this article.



(a) AIDS, Candidate Number **(b)** AIDS, Query Response Time **(c)** AIDS, GED Verification Time

(d) AIDS, Indexing Time (logarithmic) **(e)** AIDS, Candidate Number **(f)** AIDS, Query Response Time

(g) AIDS, Indexing Time **(h)** AIDS, Candidate Number **(i)** AIDS, Query Response Time

Fig. 11 Experiment results-I

8.3 Evaluating verification methods

To evaluate the extension-based verification technique, we verify the candidates returned by + Dynamic with three methods on AIDS. Besides “A*”, an algorithm “+ Extension” implementing our extension-based verification is involved. We are aware of a novel algorithm proposed lately for computing GED, namely, CSI_GED [7], which was also incorporated for comparison.

Figure 11c summarized the running time to verify the same set of candidates under varying τ’s, where the time of + Extension is used as the base for each GED threshold. We observe an improvement of + Extension over A* as much as 79%, which approaches 2 orders of magnitude when τ is small. This advantage is attributed to (1) the shrink of possible mapping space between unmatched portions of query and data graphs; and (2) the computation sharing on the matching partition between filtering and verification phases. One may concern that for a pair of graphs multiple rounds of extension-based verification can result in computational overhead. We argue that scheduling such computation requires only a little more than that of A*. Moreover, it was logged that the chance of having only one matching partition is not low, thanks to the judicious graph partitioning. For example, the percentage of candidates having only one matching partition is as high as 83% for $\tau = 1$, and 69% for $\tau = 2$, respectively. Although this percentage downgrades toward $\tau = 6$, more graphs need to conduct extension-based verification for multiple matching partitions, and the margin of response time is still large, as + Extension contributes speedups by exploring smaller search spaces. CSI_GED is a novel algorithm demonstrated to have good scalability for obtaining GED, which implements effective techniques in confining search space when GED is large. This argument is also backed by our results, where it starts slower than + Extension, and overtakes when $\tau > 4$. In short, we contend that more advanced GED algorithm can also be incorporated for verification with large τ’s, but for common and small thresholds tailored algorithm + Extension is more preferable.

8.4 Evaluating index construction

We evaluate two graph partitioning methods for index construction: (1) Random, labeled by “R”, is the basic graph partitioning method that randomly assigns vertices and edges into partitions (Algorithm 9); and (2) + Refine, labeled by

"F", is a partitioning method outlined in Algorithms 9 and 10, i.e., the complete partitioning algorithm.

Figure 11d compares the indexing time of the two algorithms in logarithmic scale. The logged time does *not* include the time of constructing index for estimating the probability that a partition is contained by a query, i.e., the index for subgraph isomorphism test, as it is reasonable to assume it is available in a graph database. We used Swift-index [20] for fast subgraph isomorphism test. Random is quite fast for all the thresholds. + Refine is more computationally demanding, typically two orders of magnitude slower than Random due to the high complexity of (1) graph partitioning optimization, and (2) partition support evaluation. Running + Dynamic on the indexes, we plot ratios of candidate number and response time in Fig. 11e, f, respectively. Together, they advise that refining random partitioning brings down candidate number by as much as 47% and response time by up to 69%.

8.5 Evaluating sample ratio

This set of experiments study the effect of sample ratio $\rho = \frac{|Q|}{|G|}$. Figure 11g–i summarize indexing time, candidate number and query response time, respectively, with varying ρ, where the best result at each GED threshold is used as the base. It can be seen that indexing time rises along with larger sample size, while candidate number and query response time exhibit slight decrease. To balance the cost and benefit of index construction, we chose $\rho = 0.4$ for subsequent experiments. Note that we did not deliberately exclude the 100 query graphs from the samples, and thus, they may appear in the samples especially when ρ is large; additionally, it can be conjectured that system performance improves if they are directly used as Q for indexing.

Hereafter, we use + Refine for offline indexing, and apply + Dynamic and + Extension for filtering and verification, respectively, to achieve the best performance.

8.6 Comparing with existing methods

This subsection compares with the state-of-the-art,

- **Pars**, labeled by "P", is our partition-based algorithm, integrating all the proposed techniques.
- **SEGOS**, labeled by "S", is an algorithm based on stars, incorporating novel indexing and search strategies [24]. We received the source code from the authors. As verification was not covered in the original evaluation, we appended A* to verify the candidates. SEGOS is parameterized by step-controlling variables k and h, set as 100 and 1000, respectively, for best performance.
- **GSimSearch**, labeled by "G", is a path-based q-gram approach for processing similarity queries [33]. The performance of q-gram-based approaches is influenced by q-gram size. For best performance, we chose $q = 4$ for AIDS, $q = 3$ for PROTEIN, and $q = 1$ for NASA.
- **BranchMix**, labeled by "M", is a hybrid method integrating branch-based and partition-based filtering techniques [34]. We implemented the algorithm, and used the A* to verify the candidates.[12] For best performance, we adopted $T = 8$ for all the datasets by default [34], which limits the largest sizes of subgraphs of query graphs for mismatching disjoint partition test.

Table 3 Index size (MB, $\tau = 6$)

Dataset	SEGOS	GSimSearch	BranchMix	Pars
AIDS	5.06	31.51	10.65	12.87
PROTEIN	0.16	2.60	0.25	0.38
NASA	11.97	8.66	12.55	14.40

Table 4 Pars index statistics ($\tau = 6$)

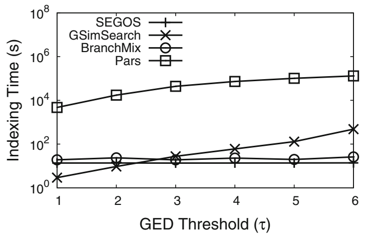
Dataset	$\lvert\mathcal{P}\rvert$	avg $\lvert I_p\rvert$
AIDS	45,263	6.60
PROTEIN	3485	1.21
NASA	46,343	5.56

Among others, κ-AT was omitted, since GSimSearch was demonstrated to outperform κ-AT under all settings [33].

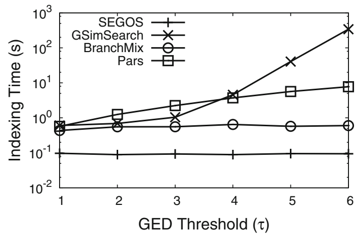
We first compare the index size. Table 3 displays the index sizes of the algorithms on three datasets for $\tau = 6$. Similar pattern is observed under other τ values. While all the algorithms exhibit small index sizes, there is no overall winner. On AIDS and PROTEIN, GSimSearch needs more space than SEGOS, BranchMix and Pars; on NASA, SEGOS, BranchMix and Pars build larger index than GSimSearch. The reason why Pars constructs a smaller index on AIDS than on NASA is that NASA possesses more large graphs. Thus, the index size of Pars is largely dependent on graph size. To get more insight of the inverted index, we list the number of distinct partitions and the average length of a postings list in Table 4. Due to judicious partitioning, the average length of postings lists is small. On PROTEIN, postings lists are shorter than the other two, because of its less number of graphs and greater diversity in substructure caused by higher vertex degrees.

Indexing time is provided in Fig. 12a–c in logarithmic scale. Pars spends more time to build index, since it involves sophisticated graph partitioning and subgraph isomorphism tests in the refine phase of index construction. It is noted that on PROTEIN, GSimSearch overtakes Pars when $\tau > 3$, due to larger density of PROTEIN graphs, and hence greater

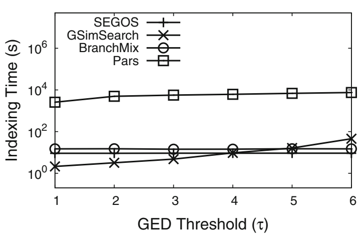
[12] The original algorithm does not come with exact verification [7].



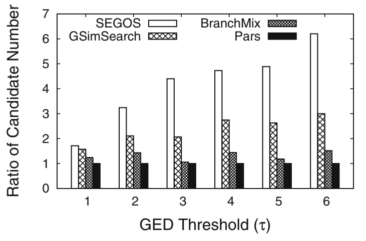
(a) AIDS, Indexing Time (logarithmic)



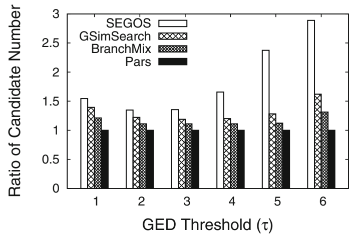
(b) PROTEIN, Indexing Time (logarithmic)



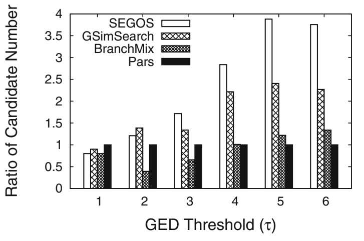
(c) NASA, Indexing Time (logarithmic)



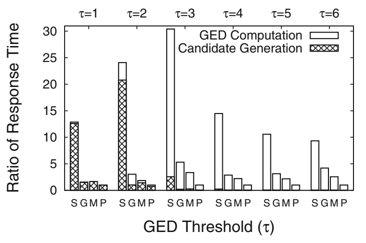
(d) AIDS, Candidate Number



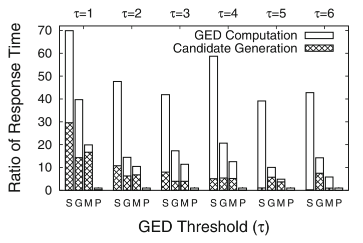
(e) PROTEIN, Candidate Number



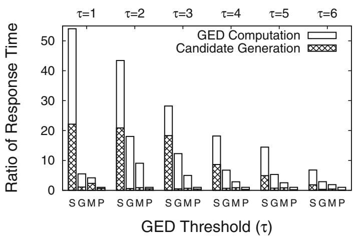
(f) NASA, Candidate Number



(g) AIDS, Query Response Time



(h) PROTEIN, Query Response Time



(i) NASA, Query Response Time

Fig. 12 Experiment results-II

difficulty in computing minimum prefix length for path-based q-grams. Among others, SEGOS and BranchMix are fairly stable since their index construction algorithms are not influenced by distance threshold for indexing.

Regarding query processing, Pars offers the best performance on both candidate size and response time in most sets of experiments, as shown in Fig. 12d–f, g–i, respectively, where the result of Pars is used the base for each GED threshold. It is read from the figures that the performance gaps between Pars and other competitors, e.g., SEGOS and BranchMix, on NASA are larger than those on AIDS. We argue that Pars is less vulnerable to large maximum vertex degrees. The numbers of candidates from SEGOS, GSimSearch and Pars are up to 114.1×, 87.0× and 53.2× that of real results (not shown), respectively. Hence, the result on response time becomes expectable. We highlight the follows: (1) Pars always demonstrates the best overall runtime performance; (2) For filtering time, GSimSearch takes more on PROTEIN, while SEGOS spends more on NASA; (3) Verification dominates the query processing stage, and GED verification on PROTEIN is more expensive than that on other datasets; (4) The margins on candidate number and response time between Pars and competitors enlarge, when τ approaches large values. We also observe that advantage of Pars is more remarkable on datasets with higher degrees like PROTEIN and NASA. For instance, when $\tau = 4$, Pars has 12.3× speedup over SEGOS on AIDS, 56.8× on PROTEIN and 19.3× on NASA. In comparison with GSimSearch, Pars is 5.1×, 42.8× and 17.5× faster, respectively on the three datasets. As to the state-of-the-art BranchMix, Pars outperforms by 2.4×, 19.6× and 8.6×, respectively. Among others, it is noted that in terms of candidate size, Pars generates a little more candidates than BranchMix for small τ's on NASA. This may be attributed to the peculiar structure of graphs in NASA, on which Pars may be less effective.

In terms of response time, however, BranchMix consumes longer time than Pars, even than GSimSearch, in the filtering phase on NASA; in addition, the verification time of Pars is much shorter than that of BranchMix, due to advanced algorithmic design. In short, the cross-method study demonstrates that Pars offers, in terms of running time, the most competent option for filtering, which is attributed to the novel partition-based filtering scheme with enhanced matching condition and mismatching partition recycle; on top of the small candidate set surviving the filtering, advanced and tailored verification procedure is carried out to ensure the best responsiveness.

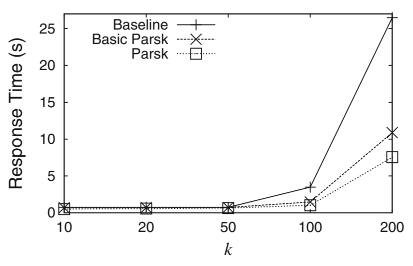
8.7 Evaluating Top-*k* search

In this set of experiments, we assess the top-*k* search algorithm by extending the partition-based filtering scheme, namely Parsk. In particular, we implemented two algorithms as described in Sect. 7:

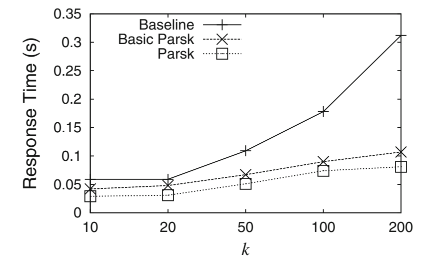
- **Baseline** is an adapted algorithm that issues threshold-based similarity searches with τ growing starting with 0, which terminates when the priority queue of answers reaches size k after a certain τ;
- **Basic Parsk** is our baseline algorithm for top-k graph similarity search leveraging partition-based filtering scheme;
- **Parsk** is an algorithm by integrating the look-ahead and computation-sharing strategies with Basic Parsk.

We first look into index construction. The space required for storing the indexes are shown in Table 5. It is read from the table that the size of the hierarchical index is mainly dependent on the dataset cardinality such that more data graphs imply more variety of partitions and longer inverted lists. Similar to the indexing performance in threshold-based similarity search, it takes time to construct the indexes, since it involves large numbers of subgraph isomorphism tests. Thanks to the one-off nature of index construction, we will see shortly this is rewarding in query processing.

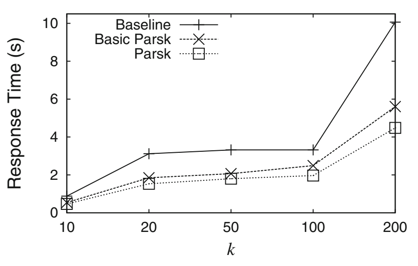
The results on query response time are plotted in Fig. 13a–c in logarithmic scale for three datasets, respectively. The general trend observed is that the query response time grows with k, but the three algorithms differ in growth ratio—the performance gap between Baseline and Parsk (and Basic



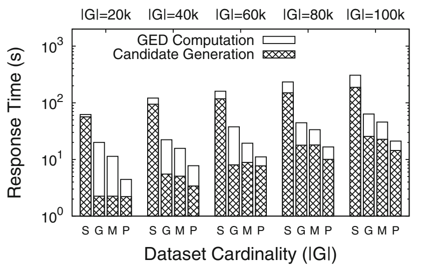
(a) AIDS, Query Response Time (logarithmic)



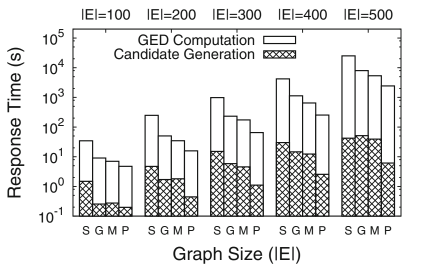
(b) PROTEIN, Query Response Time (logarithmic)



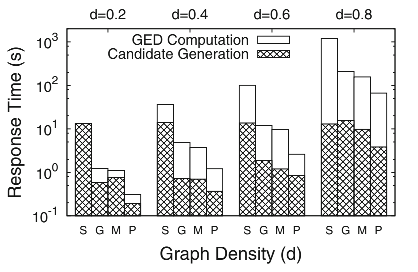
(c) NASA, Query Response Time (logarithmic)



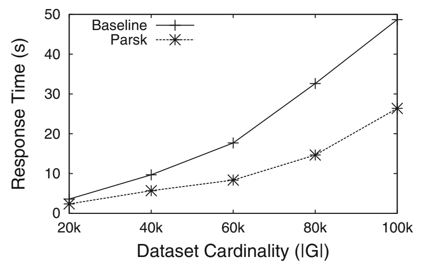
(d) Synthetic, Query Response Time (logarithmic)



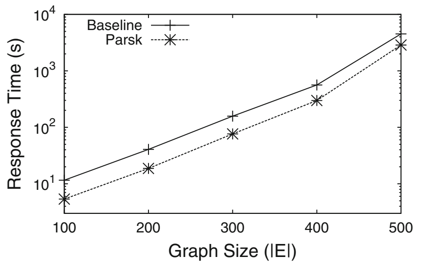
(e) Synthetic, Query Response Time (logarithmic)



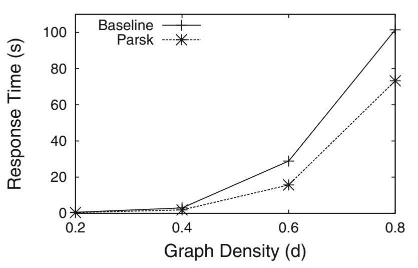
(f) Synthetic, Query Response Time (logarithmic)



(g) Synthetic, Query Response Time (logarithmic)



(h) Synthetic, Query Response Time (logarithmic)



(i) Synthetic, Query Response Time (logarithmic)

Fig. 13 Experiment results-III

Parsk) is larger toward larger τ. The figures all read that the proposed top-k similarity search framework is smarter in returning the exact answers with less computation. In particular, the gap between Baseline and Parsk is the most noteworthy on AIDS in terms of difference value, as large as 18.940, and on PROTEIN in terms of multiple, as many as 3.9×. For Baseline, we see plateaus along with the growth of k, for instance, $k \in [10, 50]$ on AIDS and $k \in [20, 100]$ on NASA. This is reasonable due to its iteration-based algorithmic design, for instance, if $k = 20$, it needs to finish 3 threshold-based similarity searches with τ equal 0, 1, 2 and 3, respectively (cf. Fig. 12d–f), which is also the case for $k = 50$. Among others, the batch-based pruning of Basic Parsk enables early stopping, and the look-ahead and computation-sharing strategies further expedite the query processing.

8.8 Evaluating scalability

All the scalability tests were conducted on synthetic data, and we first look at threshold-based setting with τ fixed as 2. To evaluate the scalability against dataset cardinality, we generated five datasets, constituted of 20–100 k graphs. Results for the 4 algorithms (cf. Sect. 8.6) are provided in Fig. 13d. The query response time grows steadily when the dataset cardinality increases. Pars has a lower starting point when dataset is small, and showcases a smaller growth ratio, with up to 18.8× speedup over SEGOS, 6.8× over GSimSearch and 4.2× over BranchMix, respectively. SEGOS consumes the most time in both filtering and verification; the filtering time of BranchMix is sometimes larger than that of GSimSearch, which may be due to the fluctuation of graph density; Pars always demonstrates the preeminent runtime performance on all five datasets.

Next, we evaluate the scalability against graph size and density on synthetic data. Each set of data graphs was of cardinality 10 k, and we randomly sampled 100 graphs from data graphs and added a random number ($[1, \tau + 1]$) of edit operations to make up the corresponding query graphs.

Five datasets with density 0.1 were generated, with average graph size ranging in [100, 500]. As shown in Fig. 13e, the query response time grows gradually with the graph size, and verification takes the majority of the total running time. Further, it takes much longer to verify the candidates of large graphs than small ones. Pars scales the best at both filtering and verification stages. This is credited to its (1) fast filtering with substantial candidate reduction, and (2) efficient verification for evaluating the candidates. On large graphs, GSimSearch spends more time on filtering, while SEGOS scales better in filtering time but becomes less effective in overall performance; BranchMix has comparable filtering performance, and expends longer time than SEGOS and GSimSearch except Pars.

Figure 13f shows the response time against graph density. Pars scales the best with density in terms of overall query response time, while SEGOS has the smallest growth ratio for filtering time. When graphs become denser, more candidates are admitted by SEGOS and GSimSearch, due to the shortcomings pointed out in Sect. 2.2. BranchMix generates less candidates than GSimSearch, but consumes longer time in filtering phase when graph density is small. By contrast, Pars exhibits good filtering and overall performance, offering up to 18.4× speedup over SEGOS, 3.4× over GSimSearch, and 2.7× over BranchMix, respectively.

Then, we assess the scalability of Baseline and Parsk (cf. Sect. 8.7) against dataset cardinality, graph size and graph density, respectively. The corresponding experiment results of Baseline and Parsk are provided in Fig. 13g–i, where the aforementioned synthetic datasets were inherited, and k was fixed as 50. From Fig. 13g, we see a steady growth of running time for both Baseline and Parsk, from 20 k graphs to 100 k graphs. Nonetheless, the gap between them is more prominent when the dataset is larger, e.g., the difference of response time is 1.2 s on 20 k graphs, and it increases to 22.3 s on 100 k graphs. Figure 13h reads a exponential growth of response time for both Baseline and Parsk. The increase is mainly attributed to the rise of complexity of verifying large graphs, which makes the results within expectation. At last, similar trend is observed in the scalability experiment against graph density, as shown in Fig. 13i. We can see that Parsk handles top-k similarity search over dense graphs better than Baseline. Moreover, the rise of graph density accounts for the upward tendency of both lines. Compared with the straightforward approach by Baseline, however, the better design of search framework and optimization techniques by Parsk reduces such effect. In short, Parsk is computationally less expensive and more scalable, when dealing with dense graphs.

Table 5 Hierarchical inverted index size (MB, $\tau_{\max} = 6$)

Dataset	AIDS	PROTEIN	NASA
Parsk	34.37	3.18	51.26

9 Related work

Research on using GED for chemical molecular analysis dates back to 1990s [30].

Structure similarity search Structure similarity search has received considerable attention recently, which is to find data graphs *approximately isomorphic* to the query graph. Since we have thoroughly reviewed the prior work for GED constraints in Sect. 2.2, here discusses other work that is also under this theme. Closure-Tree was proposed to identify top-k graphs nearly isomorphic to query [11] without precision guarantee. Incorporating the general definition of GED, Zeng et al. [29] proposed to bound GED-based structure similarity via bipartite matching of *star structures* from two graphs. It was followed by a recent effort SEGOS [24] that proposed an indexing and search paradigm based on star structures, where TA and CA are adapted to accelerate query processing. Another advance defined q-grams on graphs [23], which was inspired by the idea of q-grams on strings. It builds index by generating *tree*-based q-grams, and produces candidates against a count-filtering condition on the number of common q-grams between graphs. Seeing that limited pruning power of star structures and tree-based q-grams, GSimSearch [33] tackles the problem by utilizing *paths* as q-grams, exploiting both the matching and mismatching features. Meanwhile, Zheng et al. approached the problem by proposing *branch structures* [34]—a compromise between star structure and tree, as well as a set of GED bounding and indexing techniques. The aforementioned methods utilize fixed-size overlapping substructures for indexing, and thus, suffer from the issues summarized in Sect. 2.2. As opposed to this type of substructures, we propose a partition-based framework, and index variable-size non-overlapping partitions of data graphs.

With a focus on *representativeness* modeling, Ranu et al. [14] formulated the problem of top-k representative queries on graph databases. It finds data graphs meeting the given GED threshold whose representative power is maximized. We are also aware that *maximum common subgraph* (MCS) based similarity is investigated to find top-k results from graph databases [35]. To enhance online performance, distance and structure-preserved mapping was investigated to map graphs into vector space and find approximate top-k results [36]. We differ by developing tailored top-k search strategy leveraging a hierarchical index. Note that these are the only *two* existing work that measure structure similarity based on MCS.[13] Top-k search in a large graph [28] is based on a disparate setting and beyond our focus.

Subgraph similarity queries Subgraph similarity search is to retrieve the data graphs that approximately contain the query; most work focuses on MCS-based similarity [12,19,26]. Grafil [26] proposed the problem, where similarity was defined as the number of missing edges regarding maximum common subgraph. GrafD-index [19] deals with similarity based on maximum connected common subgraph, and it exploits the triangle inequality to develop pruning and validation rules. PRAGUE [12] developed a more efficient solution utilizing system response time under the visual query formulation and processing paradigm. Zhao et al. [33] extended path-based q-gram technique to handle *subgraph edit distance* based subgraph similarity search. Subgraph similarity matching queries were studied over single large graphs as well, [8,27] to name a few recent efforts.

As a special case when the similarity threshold is nil, subgraph similarity queries evolve into the problem of subgraph search or matching. A subgraph search query finds from a set of data graphs those containing the query graph. Practical algorithms to this problem includes the classic backtrack algorithm and a successive of advances leveraging bit vectors [21], degree reduction [22], and various indexes [20,25]. As a more difficult problem, subgraph matching finds from a single large data graph all embeddings of the query graph. A volume of literature is dedicated to this problem, e.g., [1,10,16] to name some late progress.

GED computation Another line of related research focuses on GED computation. A widely accepted exact method is based on the A^* algorithm. So far, the most well-known exact solution is attributed to an A^*-based algorithm incorporating a bipartite heuristic [17]. Our extension-based verification inherits the merit, and further conducts the search in a more efficient manner under the partition-based paradigm. Lately, a novel edge-based mapping method was proposed for computing GED through common substructure isomorphism enumeration [7]. The CSI_GED algorithm utilizes backtracking search combined with a number of heuristics, which is can be integrated as the verifier into any solutions to graph similarity searches. To render the matching process less computationally demanding, approximate methods were proposed to find suboptimal answers, e.g., [4,15]. To quickly obtain a tight search threshold in top-k similarity search queries, we may leverage some existing GED upper-bounding techniques [6,29] to restrict the search space in the beginning stage, since the real distance never exceeds the upper bound.

10 Conclusion

In this article, we have studied the problem of graph similarity search with edit distance constraints. Unlike existing solutions, we propose a framework availing a novel filtering scheme based on variable-size non-overlapping partitions. We devise a dynamic partitioning technique with enhanced matching condition and mismatching partition recycling to strengthen the pruning power. Moreover, an extension-based verification algorithm leveraging matching partitions is conceived to expedite the final verification. For index construction, a cost-aware graph partitioning method is proposed to optimize the index. In addition, the index is extended to form a hierarchical inverted index, in order to support top-k similarity queries. Based on it, tailored search procedure with look-ahead and computation-sharing strategies are devised.

[13] There is a pile of literature dedicated to subgraph similarity search based on MCS, e.g., [12,19,26].

During the research, we became aware of that certain applications may have additional *context-aware constraints*; e.g., in chemistry, an atom O may be replaced by S but not C. This can be solved by enforcing context-aware rules when computing GED with additional care, while filtering techniques are not affected and still applicable. As future work, one may come up with more pertinent filtering techniques and query processing methods to handle these constraints. Among others, the A^*-based GED algorithm is notoriously memory-consuming, due to its best-first nature. This makes it inaccessible to common PC machines nowadays. Therefore, it is of significance to develop algorithms that guarantees optimality while allowing early termination.

Acknowledgements Funding was provided by Japan Society for the Promotion of Science (Grant No. 16H01722), National Natural Science Foundation of China (Grant Nos. 61402494, 71690233), Natural Science Foundation of Hunan Province (Grant No. 2015JJ4009) and Australian Research Council (Grant Nos. DP150103071 and DP150102728).

References

1. Bi, F., Chang, L., Lin, X., Qin, L., Zhang, W.: Efficient subgraph matching by postponing Cartesian products. In: SIGMOD Conference, pp. 1199–1214 (2016)
2. Bunke, H., Allermann, G.: Inexact graph matching for structural pattern recognition. PRL **1**(4), 245–253 (1983)
3. Conte, D., Foggia, P., Sansone, C., Vento, M.: Thirty years of graph matching in pattern recognition. IJPRAI **18**(3), 265–298 (2004)
4. Fankhauser, S., Riesen, K., Bunke, H.: Speeding up graph edit distance computation through fast bipartite matching. In: GbRPR, pp. 102–111 (2011)
5. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness, 1st edn. W. H. Freeman, San Francisco (1979)
6. Gouda, K., Arafa, M., Calders, T.: Bfst_ed: a novel upper bound computation framework for the graph edit distance. In: SISAP, pp. 3–19 (2016)
7. Gouda, K., Hassaan, M.: CSI_GED: an efficient approach for graph edit similarity computation. In: ICDE, pp. 265–276 (2016)
8. Gupta, M., Gao, J., Yan, X., Cam, H., Han, J.: Top-k interesting subgraph discovery in information networks. In: ICDE, pp. 820–831 (2014)
9. Han, J., Kamber, M., Pei, J.: Data Mining: Concepts and Techniques, 3rd edn. Morgan Kaufmann, Los Altos (2011)
10. Han, W.-S., Lee, J., Lee, J.-H.: Turbo$_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases. In: SIGMOD Conference, pp. 337–348 (2013)
11. He, H., Singh, A.K.: Closure-Tree: an index structure for graph queries. In: ICDE, p. 38 (2006)
12. Jin, C., Bhowmick, S.S., Choi, B., Zhou, S.: PRAGUE: towards blending practical visual subgraph query formulation and query processing. In: ICDE, pp. 222–233 (2012)
13. Marín, R.M., Aguirre, N.F., Daza, E.E.: Graph theoretical similarity approach to compare molecular electrostatic potentials. J. Chem. Inf. Model. **48**(1), 109–118 (2008)
14. Ranu, S., Hoang, M.X., Singh, A.K.: Answering top-*k* representative queries on graph databases. In: SIGMOD Conference, pp. 1163–1174 (2014)
15. Raveaux, R., Burie, J.-C., Ogier, J.-M.: A graph matching method and a graph matching distance based on subgraph assignments. PRL **31**(5), 394–406 (2010)
16. Ren, X., Wang, J.: Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. PVLDB **8**(5), 617–628 (2015)
17. Riesen, K., Fankhauser, S., Bunke, H.: Speeding up graph edit distance computation with a bipartite heuristic. In: MLG (2007)
18. Sanfeliu, A., Fu, K.-S.: A distance measure between attributed relational graphs for pattern recognition. IEEE Trans. Syst. Man Cyber. **13**(3), 353–362 (1983)
19. Shang, H., Lin, X., Zhang, Y., Yu, J.X., Wang, W.: Connected substructure similarity search. In: SIGMOD Conference, pp. 903–914 (2010)
20. Shang, H., Zhang, Y., Lin, X., Yu, J.X.: Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. PVLDB **1**(1), 364–375 (2008)
21. Ullmann, J.R.: Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism. ACM J. Exp. Algorithmics **15**, 1–6 (2010)
22. Ullmann, J.R.: Degree reduction in labeled graph retrieval. ACM J. Exp. Algorithmics **20**, 1–3 (2015)
23. 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)
24. Wang, X., Ding, X., Tung, A.K.H., Ying, S., Jin, H.: An efficient graph indexing method. In: ICDE, pp. 210–221 (2012)
25. Yan, X., Yu, P.S., Han, J.: Graph indexing: a frequent structure-based approach. In: SIGMOD Conference, pp. 335–346 (2004)
26. Yan, X., Yu, P.S., Han, J.: Substructure similarity search in graph databases. In: SIGMOD Conference, pp. 766–777 (2005)
27. Yang, S., Han, F., Wu, Y., Yan, X.: Fast top-k search in knowledge graphs. In: ICDE (to appear) (2016)
28. Yang, Z., Fu, A.W., Liu, R.: Diversified top-*k* subgraph querying in a large graph. In: SIGMOD Conference, pp. 1167–1182 (2016)
29. 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)
30. Zhang, K., Wang, J.T.-L., Shasha, D.: On the editing distance between undirected acyclic graphs and related problems. In: CPM, pp. 395–407 (1995)
31. Zhang, S., Yang, J., Jin, W.: SAPPER: subgraph indexing and approximate matching in large graphs. PVLDB **3**(1), 1185–1194 (2010)
32. Zhao, X., Xiao, C., Lin, X., Liu, Q., Zhang, W.: A partition-based approach to structure similarity search. PVLDB **7**(3), 169–180 (2013)
33. Zhao, X., Xiao, C., Lin, X., Wang, W., Ishikawa, Y.: Efficient processing of graph similarity queries with edit distance constraints. VLDB J. **22**(6), 727–752 (2013)
34. Zheng, W., Zou, L., Lian, X., Wang, D., Zhao, D.: Efficient graph similarity search over large graph databases. IEEE Trans. Knowl. Data Eng. **27**(4), 964–978 (2015)
35. Zhu, Y., Qin, L., Yu, J.X., Cheng, H.: Finding top-*k* similar graphs in graph databases. In: EDBT, pp. 456–467 (2012)
36. Zhu, Y., Yu, J.X., Qin, L.: Leveraging graph dimensions in online graph search. PVLDB **8**(1), 85–96 (2014)