A Partition-Based Approach to Structure Similarity Search

Xiang Zhao[†] Chuan Xiao[‡] Xuemin Lin [†] [§] Qing Liu[♮] Wenjie Zhang[†]

[†] *The University of New South Wales, Australia* [‡] *Nagoya University, Japan*
[§] *Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, China* [♮] *CSIRO, Australia*
{xzhao, lxue, zhangw}@cse.unsw.edu.au chuanx@nagoya-u.jp q.liu@csiro.com

ABSTRACT

Graphs are widely used to model complex data in many applications, such as bioinformatics, chemistry, social networks, pattern recognition, etc. A fundamental and critical query primitive is to efficiently search similar structures in a large collection of graphs. This paper studies the graph similarity queries 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 or large distance thresholds. In this paper, 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 new paradigm. A candidate pruning technique and an improved graph edit distance algorithm are also developed to further boost the performance. In addition, a cost-aware graph partitioning technique is devised to optimize the index. Extensive experiments demonstrate our approach significantly outperforms existing approaches.

1. INTRODUCTION

Recent decades have witnessed a rapid proliferation of data modeled as graphs, such as chemical and biological structures, business processes and program dependencies. As a fundamental and critical query primitive, graph search, which retrieves the occurrence of a query structure in the database, is frequently issued in these application domains, and hence, has attracted extensive attention lately. Due to the existence of data inconsistency, such as erroneous data entry, natural noise, and different data representation in different sources, a recent trend is to study similarity queries.

A structure similarity search finds all data graphs from a graph collection that are similar to a given query graph. Various similarity or distance measures have been utilized to quantify the similarity between graphs, e.g., the measures based on maximum common subgraphs (MCS) [12, 16], or missing edges [19, 22]. Among them, graph edit distance (GED) stands out for its elegant property: (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. For this reason, we study structure similarity search with edit distance constraints in this paper: 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 NP-hardness of GED computation poses serious algorithmic challenges. Therefore, state-of-the-art solutions are mainly based on a *filter-and-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 [14] and path-based q-gram [21] 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. Besides q-gram features, star structure [17] was also proposed, which is exactly the same as tree-based 1-gram. Rather than count common features, [17] developed a method to compute the lower and upper bounds of GED through bipartite matching between the star representations of two graphs. The method was later equipped with a two-level index and a cascaded search strategy to find candidates [15].

We summarize the aforementioned work, i.e., (tree-based and path-based) q-grams and star 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 (2) *overlapping* – sharing vertices and/or edges in the original graphs. As a consequence, these approaches inevitably suffer from the following drawbacks: (1) They do not take full advantage of the global topological structure of the graphs and the distributions of data graphs/query workloads, and the fixing substructure size limits its selectivity, being nonadaptive to the database and queries. (2) Redundancy exists among features, hence making their filtering conditions – all of which are established in a pessimistic way to evaluate the effect of edit operations – vulnerable to large vertex degrees or large distance thresholds.

In this paper, we propose a novel filtering paradigm by dividing data graphs into *variable-size non-overlapping* partitions. We observe that such partition-based scheme is not prone to be affected by vertex degrees, and can accommodate larger distance thresholds in practice. This enables us to conduct similarity search on a wider range of applications

Proceedings of the VLDB Endowment, Vol. 7, No. 3

with larger thresholds. Another novelty is to dynamically rearrange partitions to adapt the online query by recycling and making use of the information in mismatching partitions. A filtering technique is accordingly proposed to reduce candidates, in case the partitioning of data graphs does not well fit the structural characteristics of the query. For GED evaluation, we design a verification method by extending matching partitions. Additionally, a cost model is devised to compute high-quality partitioning of data graphs for a workload of queries. The proposed techniques constitute a new graph similarity search algorithm, the superiority of which is witnessed by empirical results.

To summarize, we make the following contributions:

- We propose a novel partition-based filtering scheme for processing graph similarity search queries with edit distance constraints. To the best of our knowledge, this is among the first to use variable-size non-overlapping substructures for graph indexing and filtering.
- We design a dynamic partition filtering technique to strengthen the partition-based scheme. We devise a verification method to efficiently compute GED utilizing the matching partition between the data graph and the query. We develop a cost-aware algorithm to partition data graphs into half-edge graphs for indexing.
- We present a new framework integrating the proposed techniques, and develop an algorithm Pars implementing the framework. We conduct extensive experiments using public datasets in different application domains. The proposed algorithm is demonstrated to outperform other alternatives.

The rest of the paper is organized as follows. Section 2 presents the problem definition and the background information. Section 3 proposes a partition-based filtering paradigm. Sections 4 and 5 elaborate a dynamic partition filtering and an extension-based verification method, respectively. A cost-aware graph partitioning approach for index construction is investigated in Section 6. We provide the experimental results and analyses in Section 7. Section 8 briefs the related work, followed by conclusion in Section 9.

Note that apart from GED-based model, there is *one* existing work [23] on graph similarity search, which measures the similarity between two graphs based on MCS [1]. Based on the discussion in Appendix B of [20], we argue that GED may potentially provide richer semantics than that of MCS-based models. Thus, we adopt GED as the similarity measure in this paper.

2. PRELIMINARIES

2.1 Problem Definition

For ease of exposition, we focus on *simple* graphs, i.e., undirected graphs with neither self-loops nor multiple edges. Our approaches can be extended to directed graphs 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 a vertex v. $l_g((u,v))$ denotes the label of the edge between u and v. γ_g denotes the maximum vertex degree in g.

A *graph edit operation* is an edit operation to transform one graph to another [1, 11], 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;
- change the label of an edge.

The *graph edit distance* (GED) between g and g', denoted by GED(g, g'), is the minimum number of edit operations that transform g to g'. Graph edit distance is a *metric*. Nevertheless, computing graph edit distance between two graphs is NP-hard [17]. For brevity, we may use "edit distance" for "graph edit distance" when there is no ambiguity.

Next, we formalize the problem of graph similarity search.

PROBLEM 1 (GRAPH SIMILARITY SEARCH). *Given a data graph collection G, a query graph q, and an edit distance threshold τ, a graph similarity search finds all the data graphs whose edit distances to q do not exceed τ.*

EXAMPLE 1. *Consider in Figure 1 a data graph collection G containing g and g'. Two molecules are modeled with vertex labels representing atom symbols and edges being chemical bonds. Subscripts are added to vertices with identical labels for the purpose of differentiation, while they correspond to the same atom symbol. A graph similarity search of query graph q with $\tau = 3$ returns g' as the answer, because GED(g', q) = 3: relabel P to N, delete the edge between S and C_3, and insert an edge between N and C_3.*

In the rest of the paper, we will focus on in-memory implementation when describing algorithms.

2.2 Prior Work

Approaching the problem with sequential scan is extremely costly, because one has to not only access the whole database but also one by one conduct the NP-hard GED computations. Thus, the state-of-the-art solutions address the problem in a *filter-and-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, κ-AT algorithm [14] defines tree-based q-grams 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-ATs between the data and the query graphs is established as

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

where $\Lambda = 1 + \gamma \cdot \frac{(\gamma-1)^\kappa - 1}{\gamma - 2}$. The lower bound tends to be small, and even below zero if there is a large vertex degree in the graph or the distance threshold is high, hence rendering it useful only on sparse graphs. To relieve the issue, [21] proposed path-based q-grams, and techniques exploiting both matching and mismatching q-grams. Nonetheless, the exponential number of paths in graphs imposes a performance concern. Moreover, the inability to handle large vertex degree and distance threshold is inherited.

[1] There is more literature on subgraph similarity search based on MCS, e.g., [7, 12, 16].

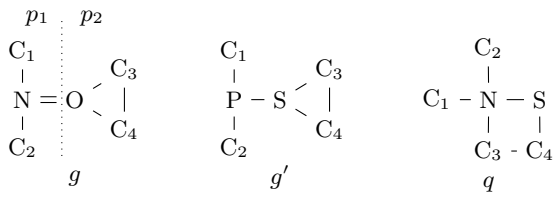


Figure 1: Sample Data and Query Graphs

A star structure [17] 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. Denote SED($g,\ q$) as the sum of pairwise distances from the bipartite matching of stars between g and q. It establishes a filtering condition on the upper bound of SED($g,\ q$) as

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

which is also proportional to the maximum vertex degree. Based on star structures, a two-level index and a cascaded search strategy were presented by SEGOS [15]. 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.

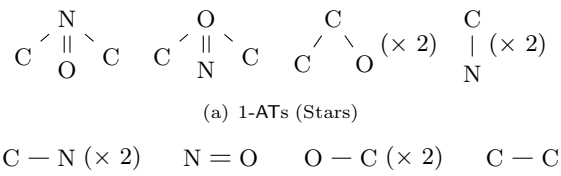
We summarize the aforementioned solutions as *fixed-size overlapping* substructure-based approaches. Intuitively, fewer candidates are usually associated with more selective features for filtering. Fixed-size features express little global structural information within the graphs and with respect to the whole database, and thus, feature selectivity is not well considered. In other words, the selectivities of frequent and infrequent features cannot be balanced 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 such disadvantages on graphs, even without large degrees or distance thresholds.

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

3. A PARTITION-BASED ALGORITHM

In this section, we propose our partition-based algorithm for graph similarity search. We first introduce the filtering principle, and then detail an algorithmic framework realizing the new filtering paradigm.

3.1 Partition-based Filtering Scheme



(a) 1-ATs (Stars)

C — N (× 2) N = O O — C (× 2) C — C

(b) Path-based 1-grams

Figure 2: Fixed-size Substructures

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

Example 3. *Consider graphs g and q in Figure 1, and $\tau = 1$. We divide g into two partitions p_1 and p_2. It can be seen that neither partitions are contained by q. Since an edit operation can occur in only one of the two partition, at least two edit operations are required to make them not contained by q. Thus, g does not satisfy the query constraint. Recall Example 2 that all existing solutions take g as q's candidate.*

The example shows the possibility of filtering data graphs by partitioning data graphs 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 a *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 $(u, \cdot)$.*

Definition 2 (half-edge graph). *A half-edge graph g is a labeled graph, denoted by a triple (V_g, E_g, l_g), where V_g is a set of vertices, $E_g \subseteq V_g \times V_g \cup V_g \times \{\cdot\}$, and l_g is a labeling function that assigns labels to vertices and 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 *$f : V_g \to V_{g'}$ such that (1) $\forall u \in V_g$, $f(u) \in V_{g'} \wedge l_g(u) = l_{g'}(f(u))$; (2) $\forall (u, v) \in E_g$, $(f(u), f(v)) \in E_{g'} \wedge l_g((u, v)) = l_{g'}((f(u),\ f(v)))$; and (3) $\forall (u, \cdot) \in E_g$, $(f(u), w) \in E_{g'} \wedge l_g((u, \cdot)) = l_{g'}((f(u), w))$, $w \in V_{g'} \setminus f(V_g)$.*

If g is half-edge subgraph isomorphic to g', we say g is a *half-edge subgraph* of g', or g is *contained* by g'. It is immediate that half-edge subgraph isomorphism test is at least as hard as subgraph isomorphism test (NP-complete [4]). Hereafter, we shorten "half-edge subgraph isomorphism" to "subgraph isomorphism" when the context is clear.

Definition 4 (graph partitioning). *A partitioning of a graph g is a division of the 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., $P(g) = \{\, p_i \mid \cup_i p_i = V_g \cup E_g \wedge p_i \cap p_j = \emptyset,\ \forall i,\ j,\ i \neq j \,\}$, where each p_i is a half-edge graph, called a* partition *of g* [2].

Example 4. *Consider graph g' in Figure 1. Figure 3 depicts one partitioning $P(g') = \{\, p'_1,\ p'_2 \,\}$ among many others, where p'_1 and p'_2 are two half-edge graphs with half-edges.*

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

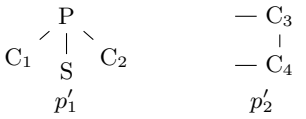


Figure 3: Example of Partitioning of g' in Figure 1

Next, we state our partition-based filtering principle.

THEOREM 1 (PARTITION-BASED FILTERING PRINCIPLE). *Consider a query 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 is subgraph isomorphic to q.*

PROOF. See Appendix A of [20]. □

We call a partition a *matching* partition if it is half-edge subgraph isomorphic to the query, or otherwise a *mismatching* partition. It is also of interest to see that given a data graph g partitioned into $\tau+1$ half-edge graphs, the filtering principle can be extended to all thresholds no larger than τ.

COROLLARY 1. *Consider a query q, a data graph g and its $\tau+1$ partitions. If $GED(g,q) \leq \tau' \leq \tau$, at least $\tau+1-\tau'$ partitions are subgraph isomorphic to q.*

Due to Corollary 1, we are able to build an index offline with a pre-defined $\tau_{\max}$, which works for all thresholds τ no larger than $\tau_{\max}$. We focus on the $\tau = \tau_{\max}$ case hereafter.

3.2 Graph Similarity Search Algorithm

In light of Theorem 1, we propose a partition-based similarity search framework Pars. It encompasses two stages – indexing (Algorithm 1) and query processing (Algorithm 2). In the indexing stage, which can be done offline, it takes as input a graph database G and an edit distance threshold τ, and constructs an inverted index. For each data graph g, it first divides g into $\tau+1$ partitions by calling PartitionGraph (Line 2, to be introduced in Section 6). Then, for each partition, it inserts g's identifier into the corresponding postings list of the partition (Lines 3 – 4).

In the online query processing stage, Algorithm 2 receives a query q, and starts probing 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, 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 (resp. label) filtering tests whether the difference exceeds τ between the data graph and the query in terms of vertex and edge numbers (resp. numbers of vertex and edge relabeling). 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 – 8). Finally, candidates are sent to GraphEditDistance, and results are returned in R (Line 9).

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 $D_p = \{ g \mid p \sqsubseteq g, g \in G \}$, $p \in \mathcal{P}$. We analyze the overall cost of processing a query:

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

where (1) t_s is the average running time of a subgraph isomorphism test; (2) t_m is the running time of retrieving and merging the postings lists of the matching partitions; and (3) t_d is the average running time of a GED computation.

Since the postings lists are usually short due to judicious graph partitioning (to be discussed in Section 6), subgraph isomorphism tests and GED computations play the major role. Thanks to recent advances, subgraph isomorphism test can be done efficiently on small graphs [13] and even large sparse graphs (with hundreds of distinct labels and up to millions of vertices) [5]. Our empirical study also demonstrates that subgraph isomorphism test is on average three orders of magnitude faster than GED computation. Therefore, we argue that the major factor of the overall cost lies in GED computation, 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 the indexed features. A matching feature [3] is prone to produce many candidates if its postings lists 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 the feature-based inverted index. We are able to choose the features reflecting the global structural information within data graphs and database, and thus to obtain statistically more selective features than the previous approaches. Furthermore, partition-based features distinguish from those utilized in previous approaches in that the partitions are non-overlapping. This

Algorithm 1: ParsIndex (R, τ)

```
   Input  : G is a collection of data graphs; τ is an edit
            distance threshold.
   Output : An inverted index I, initialized as ∅.
1  foreach g ∈ G do
2      P_g ← PartitionGraph (g);
3      foreach p ∈ P_g do
4          I_p ← I_p ∪ { g };
5  return I
```

Algorithm 2: ParsQuery (q, I, τ)

```
   Input  : q is a query graph; I is an inverted index built on G;
            τ is an edit distance threshold.
   Output : R = { g | GED(g, q) ≤ τ, g ∈ G }.
1  M ← empty map from graph identifier to boolean;
2  foreach p in I do
3      if SubgraphIsomorphism (p, q, ∅) then
4          foreach g in I_p such that M[g] is not initialized do
5              if SizeFiltering (g, q) ∧ LabelFiltering (g, q) then
6                  M[g] ← true ;        /* find a candidate */
7              else
8                  M[g] ← false ;  /* pruned by size or label
                   filtering */
9  R ← GraphEditDistance (q, M);
10 return R
```

[3] E.g., for Pars, a partition contained by the query; for κ-AT and GSimSearch, a q-gram appearing in the query's q-gram multiset.

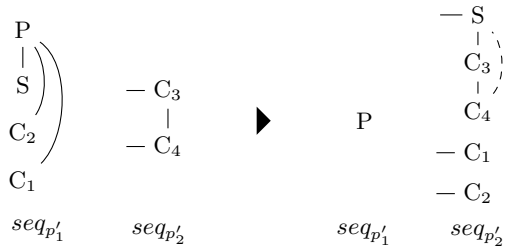


Figure 4: Example of QISequences.

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 algorithm does not suffer from the drawback of loose bounds when handling large thresholds and data graphs/queries with large degree vertices.

Before delving into the graph partitioning algorithm, we will first exploit the optimizations to reduce candidates on top of the partition-based filtering (Section 4), and discuss efficient verification of candidates (Section 5).

4. DYNAMIC PARTITION FILTERING

We start with an illustrating example to show the idea of dynamic partition filtering.

EXAMPLE 5. *Consider in Figure 1 data graph g' and query graph q, and $\tau = 1$. Assume we have partitioned g' to p'_1 and p'_2 in Figure 3. p'_1 is not contained by q but p'_2 is, making g' a candidate. However, if we adjust the partitioning by moving the vertex S from p'_1 to p'_2, neither p'_1 nor p'_2 will be contained by q, hence disqualifying g' being a candidate.*

This example evidences the chance of adjusting the partitions according an online query so that the pruning power of partition-based filtering is enhanced. This section conceives a novel filtering technique to exploit the observation, and we integrate the technique into the subgraph isomorphism test. Next, we first adapts a graph encoding technique QISequence for efficient half-edge subgraph isomorphism test, based on which a dynamic partition filtering will be presented.

4.1 Half-edge Subgraph Isomorphism Test

QISequence [13] is a graph encoding technique originally proposed for efficient (non-half-edge) subgraph isomorphism test. We extend it to support the half-edge case. The QISequence of a partition p is a regular expression $seq_p = [[v_i e^*_{ij}]^{|V_p|}]$ encoded based on the spanning trees of p's connected components. For all $i > j$, e_{ij} encodes (1) sEdge – the *spanning edge* between v_i and v_j in the spanning tree; (2) bEdge – the *backward edges* between v_i and v_j in p but not in the spanning tree; (3) hEdge – the *half-edges* incident to v_i. For the first term of each connected component, sEdge equals nil. For ease of exposition, we assume p has only one connected component [4]. To generate the QISequence of p, we start with an empty sequence at the root of a spanning tree. Then, vertices $v_i \in V_p$ are appended to QISequence in the order of the spanning tree, each along with a spanning edge, as well as possible backward edges and half-edges.

[4]For multiple connected components, sequences are generated for each component and concatenated as QISequence.

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

Input : p is a partition; q is a query graph; $\mathcal{F}$ is a mapping vector.
Output : A boolean indicating whether $p \sqsubseteq q$.

```
1 if |F| = |V_p| then
2     return true
3 v ← next vertex in seq_p;
4 U ← { u | u ∈ FindValidCandidates(v, seq_p, q, F) };
5 foreach u ∈ U do
6     F' ← F ∪ { v → u };
7     if BasicSubgraphIsomorphism (p, q, F') then
8         return true
9 return false
```

EXAMPLE 6. *Consider partition p'_1 in Figure 3. Based on a spanning trees rooted at P, the sequence $seq_{p'_1}$ of p'_1 is shown in the leftmost of Figure 4, where solid lines represent spanning edges and half-edges, and dashed lines denote backward edges.*

Algorithm 3 tests if 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 $\mathcal{F}$ between p and q in a depth-first search. For the current vertex v of p, if $seq_p[v]$ is the first term of a connected component with sEdge $= nil$, it finds candidate vertices from all unmapped vertices in V_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. These are realized by FindValidCandidates (omitted, Line 4). 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 previous vertex of p 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, indicating p is not subgraph isomorphic to q.

Correctness and Complexity Analysis. It can be verified that if there exits a half-edge subgraph isomorphism from p to q, Algorithm 3 must find it, and hence, its correctness follows. The worst case time complexity remains the same as traditional subgraph isomorphism: $O((\gamma_p \cdot \gamma_p)^{|V_p|})$.

4.2 Recycling Mismatching Partitions

We call $|\mathcal{F}|$, the cardinality of the mapping from p to q, the *depth* of the mapping $\mathcal{F}$. Among all the mappings explored by the algorithm, there is a *maximum depth* $d_{\max}$. A full mapping is found if and only if $d_{\max}$ equals $|V_p|$. Contrarily, if no full mapping is found, it implies that the vertices, which are not included in the mapping that yields $d_{\max}$, make p not contained by q. In other words, we could have allocated less vertices to p. We show how to recycle these vertices and append to other partitions, starting with an example.

EXAMPLE 7. *Consider data graph g', the query q in Figure 1, the partitioning of g' in Figure 3, and $\tau = 1$. We depict the QISequences of the two partitions in Figure 4. We first conduct subgraph isomorphism test from p'_1 to q, and no mapping is found for the first vertex P. Thus, $d_{\max} = 0$ for p'_1. Then, we conduct subgraph isomorphism test from p'_2 to q, and observe that p'_2 has a full mapping, and include g' as a*

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

Input : p is a partition; q is a query graph; $\mathcal{F}$ is a mapping vector.
Output : A boolean indicating whether $p \sqsubseteq q$.

```
1  if d_max < |F| then d_max ← |F| if |F| = |V_p| then
2      return true
3  v ← next vertex in seq_p;
4  U ← { u | u ∈ FindValidCandidates(v, seq_p, q, F) };
5  foreach u ∈ U do
6      F' ← F ∪ { v → u };
7      if RecyclingSubgraphIsomorphism (p, q, F') then
8          return true
9  if this is the outmost call then
10     foreach g in I_p such that M[g] is not initialized do
11         foreach v_i ∈ seq_p, i > d_max + 1 do
12             add v_i and its incident edges in g into Δ_g;
13 return false
```

Algorithm 5: GraphEditDistance $(g,\ q)$

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  O ← order the vertices of g;
2  F ← ∅, Q.push(F);
3  while Q ≠ ∅ do
4      F ← Q.pop();
5      if |F| = |V_g| then
6          return g(F)
7      u ← next unmapped vertex in V_g as per O;
8      foreach v ∈ V_q such that v ∉ F and
       |deg(u) − deg(v)| ≤ τ or a dummy vertex do
9          F ← F ∪ { u → v };
10         g(F) ← ExistingDistance(F);
11         h(F) ← EstimateDistance(F);
12         if f(F) = g(F) + h(F) ≤ τ then Q.push(F)
13 return τ + 1
```

candidate. However, after testing p'_1, if we recycle S, C_1, C_2 [5], and incident edges from p'_1, and append to p'_2, the QISequence of p'_2 becomes as shown in the rightmost of Figure 4. The new p'_2 is not contained by q, and thus, g' is no longer a candidate.

The basic idea of dynamic partition filtering is to leverage the mismatching partition and to dynamically add, if possible, additional vertices and edges to a partition tested to be contained by the query. Algorithm 4 implements the subgraph isomorphism test equipped with the dynamic partition filtering. $d_{\max}$ is initialized to 0 in the first call. If the algorithm returns **false** in the outmost call, the maximum depth $d_{\max}$ advises that the subgraph induced by the first $d_{\max}+1$ vertices is enough to prevent this partition from matching. As a byproduct of the subgraph isomorphism test for future use, for every data graph g having p as its partition, we respectively recycle the vertices $v_i \in seq_p, i > d_{\max}+1$ as well as their incident edges in g.

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, g becomes a candidate and is verified by GED computation. Note that if the new subgraph isomorphism test fails, the vertices and edges beyond $d_{\max}+1$ can be recycled again.

Correctness and Complexity Analysis. It can be verified Algorithm 4 correctly compute the containment relation between p and q, and the maximum mapping depth. In addition to half-edge subgraph isomorphism test, $O((|V_p| - d_{\max}-1)\cdot\delta_p)$ effort is required to collect the unused subgraph of p, where δ_p is the average vertex degree of p.

5. VERIFICATION

In this section, we present an efficient algorithm that advises whether a candidate is a result. Since for each candidate, its matching partitions have been identified through index probing, the partitions can be collected to expedite the verification. We first review a state-of-the-art GED computation algorithm, followed by the speed-up on top of it.

5.1 Graph Edit Distance Computation

[5]Note that we have to leave P in p'_1 to make $p'_1 \not\sqsubseteq q$.

The most widely used algorithm to compute GED is based on A* [10], 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. For $h(\mathcal{F})$ in weighted graphs, [3] proposes an estimation via bipartite matching. In unweighed case, it becomes exactly the numbers of vertex and edge relabeling between the remaining parts of g and q, which can be done in $O(|V_g| + |V_q|)$.

We encapsulate the details in Algorithm 5. 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 [10]. 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 4). 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 (omitted) and EstimateDistance (omitted) to calculate the values of $g(\mathcal{F})$ and $h(\mathcal{F})$, respectively. Only if $f(\mathcal{F}) \leq \tau$ is the state inserted into the queue (Lines 9 – 12).

The search space of Algorithm 5 is exponential in the number of vertices. Next, we present our improvement.

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

Algorithm 6: ExtensionBasedDistance $(g,\ q,\ p,\ \mathcal{F})$

```
1 while F ≠ ∅ do
2     distance ← GraphEditDistance(g, q, F);
3     if distance ≤ τ then
4         return distance
5     else F ← EnumerateNextMapping(p, q)
6 return τ + 1
```

Algorithm 7: Replacement of Lines 1 – 2 of Algorithm 5

```
1 g(F) ← ExistingDistance(F) ;        /* F is a subgraph
  isomorphic mapping of p in q */
2 h(F) ← EstimateDistance(F);
3 if f(F) = g(F) + h(F) ≤ τ then
4     O ← order the vertices in V_g \ V_p ; /* p is one and only
      matching partition of g */
5     Q.push(F);
```

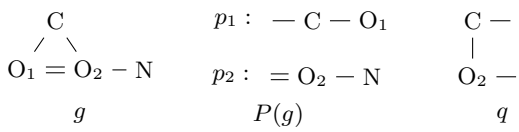


Figure 5: Example of Extension-based Verification

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 5. 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 matching partition. However, it is not easy to share the computation among different runs of the verification. In order to strike a balance, we choose to conduct the extension-based verification if g has *only one* matching partition; otherwise, we use the traditional A* verification. Our experiment (Section 7.3) shows that more than half candidates have only one matching partition when $\tau \in [1, 4]$.

THEOREM 2 (CORRECTNESS OF ALGORITHM 6). *Extension-based verification correctly computes the complete set of results over the candidates having only one matching partition.*

PROOF. See Appendix A of [20]. □

Algorithm 6 outlines the extension-based verification. It takes as input a data graph g, a query q, the only matching partition p, and the vertex mapping $\mathcal{F}$ obtained from 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 2 is no larger than τ, it returns the distance immediately; otherwise, it proceeds with the next mapping until all mappings are attempted. In each run of Algorithm 5, we let it take as input the mapping $\mathcal{F}$, and modify Lines 1 – 2 as per Algorithm 7. $g(\mathcal{F})$ 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 by the A* algorithm.

EXAMPLE 8. *Consider a data graph g with its two partitions and a query graph q shown in Figure 5, 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 $-C-O_2$. Eventually, we can verify g is not an answer since $GED(g,\ q) = 2$.*

Correctness and Complexity Analysis. The correctness of Algorithm 6 is guaranteed by Theorem 2. The worst case complexity is in $O((|V_q| \cdot (|V_g| + |E_g| + \gamma_g))^{|V_g|})$.

We remark that the search space of our solution is usually much smaller than that of Algorithm 5, as demonstrated by the empirical result in Section 7.3. By fixing the matching partition p to $\mathcal{F}(p)$, we only match an unmapped vertex in $g \setminus p$ to a vertex in $q \setminus \mathcal{F}(p)$; if the matching partition has more embeddings in q, the cost of locating other embeddings is also much smaller via subgraph isomorphism. Therefore, the proposed solution effectively shrink the search space, and share the computation between verification and filtering phases. To integrate Algorithm 6 into Algorithm 2, we need a counter instead of a boolean state to record candidates. Whenever the index probing is done, the data graphs are (1) to be verified in an extension-based fashion if the counters equal to 1; (2) to be verified by the traditional A* algorithm if the counters exceed 1; or (3) not to become candidates if the counters equal to 0.

6. COST-AWARE GRAPH PARTITION

In this section, we investigate the graph partitioning method for index construction. 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 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., $\phi_p = \frac{|\{\, q \mid p \sqsubseteq q \wedge q \in Q \,\}|}{|Q|}$. 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 candidate 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 9. *Consider $\tau = 1$, the data graph g in Figure 5, and the three graphs in Figure 1 as Q. A partitioning $P(g)$ is shown in Figure 5. Testing p_1 against Q confirms that no*

Algorithm 8: RandomPartition $(g,\ \tau)$

Input : g is a data graph; τ is an edit distance threshold.
Output : A graph partitioning P, initialized as $\emptyset$.

```
1  M ← empty map from vertex identifier to boolean ;
   /* record whether a vertex has been considered */
2  for i ∈ [1, τ + 1] do
3      randomly choose a vertex v ∈ V_g such that M[v] = false;
4      p_i ← ({ v }, ∅, { l_v });
5      M[v] ← true;
6  while ∃ a vertex v such that M[v] = false do
7      foreach p_i ∈ P do
8          u ← ChooseVertexToExpand (p_i);
9          ExpandInducedSubgraph (p_i, u);
10 while ∃ an edge (u, v) ∈ E_g with end vertices in different
   partitions do
11     randomly assign e to either p_u or p_v ;   /* half-edges */
12 return P
```

Algorithm 9: RefinePartition $(P,\ Q)$

Input : P is a graph partitioning; Q is a set of query graphs.
Output : P is an optimized graph partitioning.

```
1  c_g ← ComputeSupport (P, Q), updated ← true;
2  while updated = true do
3      c_min ← c_g;
4      foreach (u, v) ∈ E_g do
5          P' ← P;
6          p'_u ← ShrinkInducedSubgraph(p'_u, u);
7          p'_v ← ExpandInducedSubgraph(p'_v, u);
8          randomly assign remaining edges between p'_u and p'_v;
9          c'_g ← ComputeSupport(P', Q);
10         if c'_g < c_min then
11             P_min ← P', c_min ← c'_g;
12     if c_min < c_g then P ← P_min, c_g ← c_min else
       updated ← false
13 return P
```

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, 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 (Section 7.5).

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

6.2 A Practical Partitioning Algorithm

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

PROBLEM 2 (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 [6]. 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.

Algorithm 8 presents the pseudocode of the random partitioning phase of our algorithm. It 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 with 1-hop by ChooseVertexToExpand (omitted): randomly select a vertex $v \in V_{p_i}$ and include another vertex u, which has not been assigned to any partitions, and its edges connected to the vertices 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 assigned to different partitions, randomly to either the partition containing u or v as half-edges.

In the refine phase, we take the opportunity to improve the quality of the initial partition, as shown in Algorithm 9. 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 6 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 7, it adds u and edges between u and vertices in p'_v. Afterwards, 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 9. 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 round 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 1 and 9, respectively, we can conduct subgraph isomorphism test to collect partitions' support in Q, fulfilled by ComputeSupport (omitted).

Correctness and Complexity Analysis. Immediate is that Algorithms 8 and 9 compute a graph partitioning conforming to Definition 4. For Algorithm 8, it takes $O(V + E)$ time to assign vertices and edges. The complexity of Algorithm 9 is mostly determined by ComputeSupport, which carries out subgraph isomorphism tests from the partitions to Q. In each iteration of the refinement, we need to conduct $|E|$ rounds of ComputeSupport, through which the supports of two newly constructed partitions are re-evaluated.

7. EXPERIMENTS

This section reports experimental results and analyses.

[6] 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 already.

Table 1: Dataset Statistics

Dataset	$\|G\|$	avg $\|V\|/\|E\|$	$\|l_V\|/\|l_E\|$	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

7.1 Experiment Setup

We conducted experiments on public real datasets:

- **AIDS** is an antivirus screen compound dataset from the Developmental Therapeutics Program at NCI/NIH [7]. It contains 42,687 chemical compound structures.
- **PROTEIN** is a protein database from the Protein Data Bank [8], constituted of 600 protein structures. Vertices represent 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 [9], including 36,790 graphs. We randomly assigned 10 vertex labels to the graphs, as the original graphs are nearly of unique vertex labels.

Table 1 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. Thus, the queries are of similar data distribution 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 [10]. 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 were 2 and 1, respectively.

Experiments were conducted on a machine of Quad-Core AMD Opteron Processor 8378@800MHz with 96G RAM [11], running Ubuntu 10.04 LTS. All the algorithms were implemented in C++, and ran in main memory. We evaluated our solution with identical thresholds at indexing and query processing stages, i.e., $\tau = \tau_{max}$. We measured (1) index size; (2) indexing time; (3) number of candidates that need GED computation; and (4) query response time, including candidate generation and GED computation. Candidate number and running time are reported on the basis of 100 queries.

7.2 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, and "+ Dynamic" to denote the implementation of integrating Basic Partition with dynamic partition filtering.

Figure 6(a) shows the candidate number on AIDS. The candidates returned by both methods increase with the growth of τ, and the gap is more remarkable when τ is large. The number of real results is also shown for reference. The margin is substantial, and when $\tau = 1$, + Dynamic provides a reduction over Basic Partition by 51%. To reflect the filtering effect on response time, we appended the basic A* algorithm (denoted "A*", Algorithm 5) to verify the candidates.The query response time is plotted in Figure 6(b). "BP" and "AD" are short for Basic Partition and + Dynamic, respectively. The filtering time of + Dynamic is greater than Basic Partition; whereas, as an immediate consequence of less candidate number, the overall response time of + Dynamic is smaller by up to 64% among all the thresholds. Thus, dynamic partition filtering needs more computation in filtering but improves the overall runtime performance in return.

7.3 Evaluating Verification Methods

To evaluate the extension-based verification technique, we verify the candidates returned by + Dynamic with two methods on AIDS. Besides "A*", an algorithm "+ Extension" implementing our extension-based verification is involved.

Figure 6(c) reports the running time to verify the same set of candidates under different τ's. We observe an improvement of + Extension over A* as much as 76%. This advantage is attributed to (1) the shrink of possible mapping space between unmatched portions of query and data graphs; (2) the computation sharing on the matching partition between filtering and verification phases. To further validate its effectiveness, we logged how often + Extension is triggered. The percentages of the candidates having only one matching partition are 86%, 71%, 64%, 51%, 37%, 25% for $\tau \in [1, 6]$, respectively. Thus, the chance of conducting + Extension is high, especially when τ is small. The drop is intuitive, since the larger τ is, the more partitions there are for each graph, hence with the smaller each partition and the greater chance of being contained by queries. Although the ratio downgrades towards $\tau = 6$, the margin of response time is still great, as + Extension contributes speedups by exploring smaller search spaces.

7.4 Evaluating Index Construction

We evaluate two graph partitioning methods for index construction: (1) Random, labeled by "RD", is the basic graph partitioning method that randomly assigns vertices and edges into partitions (Algorithm 8); and (2) + Refine, labeled by "RF", is a partitioning method outlined in Algorithms 8 and 9, i.e., the complete partitioning algorithm.

Figure 6(d) compares the indexing time of the two algorithms. 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 [13] 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 candidate number and response time in Figures 6(e) and 6(f), respectively. Together, they advise that refining random partitioning brings down candidate number by as much as 47%, and thus, response time by up to 69%.

7.5 Evaluating Sample Ratio

This set of experiments study the effect of sample ratio $\rho = \frac{|Q|}{|G|}$. Figures 6(g) – 6(i) show the indexing time, the

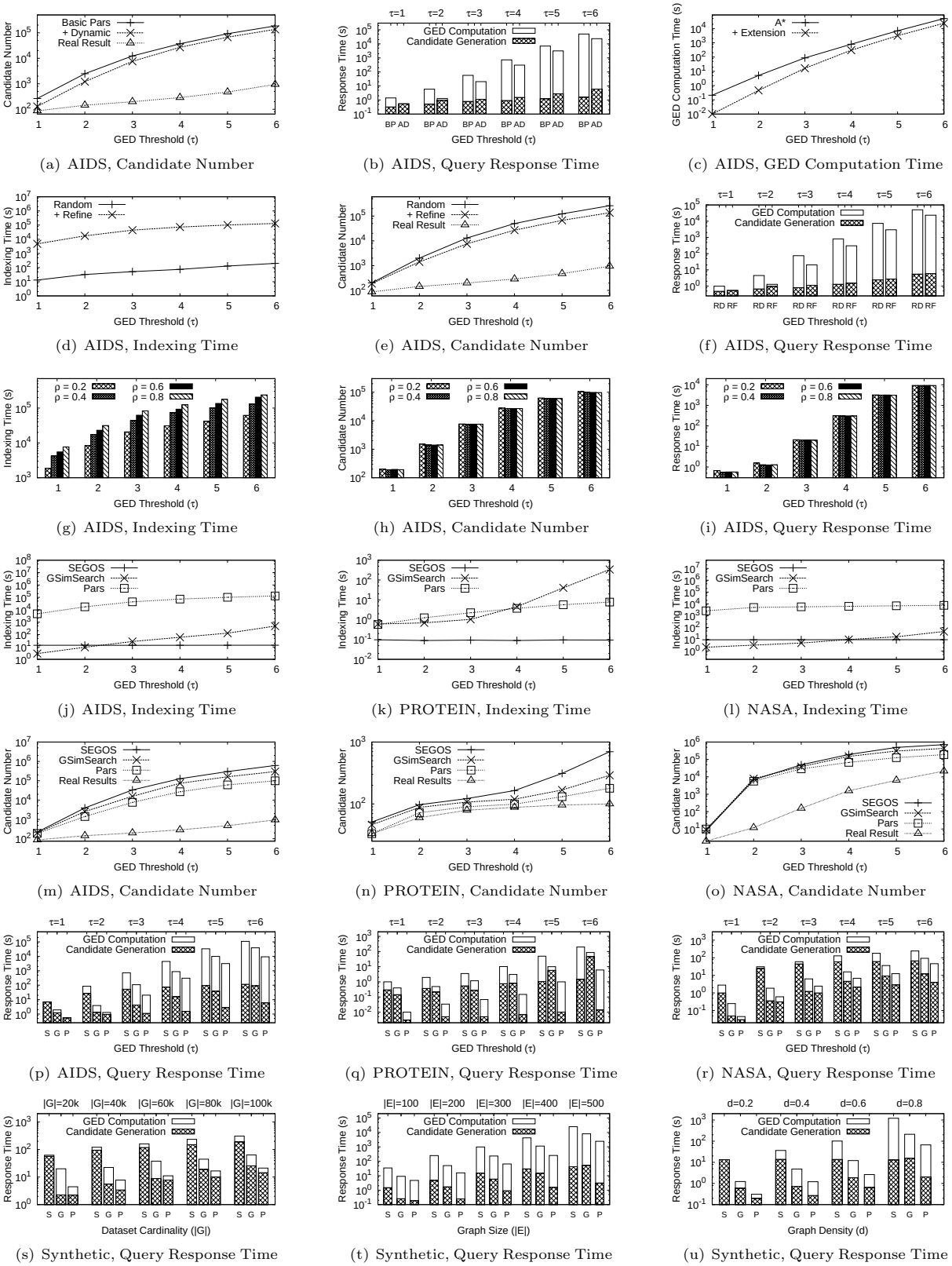
[7] http://dtp.nci.nih.gov/docs/aids/aids_data.html

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

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

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

[11] 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.



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

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

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

(j) AIDS, Indexing Time (k) PROTEIN, Indexing Time (l) NASA, Indexing Time

(m) AIDS, Candidate Number (n) PROTEIN, Candidate Number (o) NASA, Candidate Number

(p) AIDS, Query Response Time (q) PROTEIN, Query Response Time (r) NASA, Query Response Time

(s) Synthetic, Query Response Time (t) Synthetic, Query Response Time (u) Synthetic, Query Response Time

Figure 6: Experiment Results

Table 2: Index Size (MB, $\tau = 6$)

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

Table 3: Pars Index Statistics ($\tau = 6$)

Dataset	$\lvert\mathcal{P}\rvert$	avg $\lvert I_p\rvert$
AIDS	45,263	6.60
PROTEIN	3,485	1.21
NASA	46,343	5.56

candidate number and the query response time, respectively, with varying ρ. 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. We remark that system performance improves if we directly use the query graphs as Q for indexing. Hereafter, we use + Refine for indexing, and apply + Dynamic and + Extension for filtering and verification, respectively, to achieve the best performance.

7.6 Comparing with Existing Methods

This subsection compares the proposed method with the state-of-the-art solutions, involving:

- **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 [15]. 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 1,000, respectively, for best performance.
- **GSimSearch**, labeled by "G", is a path-based q-gram approach for processing similarity queries [21]. 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. κ-AT was omitted, since GSimSearch was demonstrated to outperform κ-AT under all settings.

We first compare the index size. Table 2 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 and Pars; on NASA, SEGOS 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 3. Due to judicious partitioning, the average lengths of posting lists are small. On PROTEIN, postings lists are shorter than the other two, because of its less number of graphs and diversity in substructure caused by higher degree.

Indexing time is provided in Figures 6(j) – 6(l). Pars spends more time to build index, since it involves complex graph partitioning and subgraph isomorphism tests in the refine phase of index construction. We note that on PROTEIN, GSimSearch overtakes Pars when $\tau > 3$, due to larger density of PROTEIN graphs, and hence greater difficulty in computing minimum prefix length for path-based q-grams.

Regarding query processing, Pars offers the best performance on both candidate size and response time, as shown in Figures 6(m) – 6(o) and 6(p) – 6(r), respectively. The gaps between Pars and other competitors 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 114.1x, 87.0x and 53.2x that of real results, 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 phase, and GED computation on PROTEIN is more expensive than 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 6.1x speedup over SEGOS on AIDS, 56.7x on PROTEIN and 15.3x on NASA. In comparison with GSimSearch, Pars is 2.9x, 42.6x and 7.1x faster, respectively on the three datasets.

7.7 Evaluating Scalability

All the scalability tests were conducted on synthetic data, and we fixed τ as 2. To evaluate the scalability against dataset cardinality, we generated five datasets, constituted of 20k – 100k graphs. Results are provided in Figure 6(s). 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 18.5x speedup over SEGOS and 6.6x over GSimSearch.

Next, we evaluate the scalability against graph size and density on synthetic data. Each set of data graphs was of cardinality 10k, 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 Figure 6(t), the query response time grows gradually with the graph size. 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 time.

Figure 6(u) 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 dense, more candidates are admitted by SEGOS and GSimSearch, due to the shortcomings we discussed in Section 2.2. Pars exhibits good filtering and overall performance, offering up to 18.2x speedup over SEGOS and 3.2x over GSimSearch.

8. RELATED WORK

Structure similarity search has received considerable attention recently. Closure-Tree was proposed to identify top-k graphs nearly isomorphic to query [6]. The notion of star

structures [17] were proposed, and the edit distance constraint can be converted to lower and upper bounds of star structure distance via bipartite matching. It was followed by a recent effort SEGOS [15] that proposed an indexing and search paradigm based on star structures. Another advance defined q-grams on graphs [14], 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. Similarly, GSimSearch [21] approaches the problem by utilizing paths as q-grams, exploiting both the matching and mismatching features. These approaches utilize fixed-size overlapping substructures for indexing, and thus, suffer from the issues summarized in Section 2.2. As opposed to this type of substructures, we propose to index the variable-size non-overlapping partitions of data graphs.

Subgraph similarity search is to retrieve the data graphs that approximately contain the query; most work focuses on MCS-based similarity [7, 12, 16]. Grafil [16] proposed the problem, where similarity was defined as the number of missing edges regarding maximum common subgraph. GrafD-index [12] dealt with similarity based on maximum connected common subgraph, and it exploits the triangle inequality to develop pruning and validation rules. PRAGUE [7] developed a more efficient solution utilizing system response time under the visual query formulation and processing paradigm. Subgraph similarity queries were studied over single large graphs as well, [9, 22] to name a few recent efforts.

Research on using GED for chemical molecular analysis dates back to 1990s [18]. To compute GED, so far the fastest exact solution is attributed to an A*-based algorithm incorporating a bipartite heuristic [10]. Our extension-based verification inherits the merit, and further conducts the search in a more efficient manner under the partition-based paradigm. To render it less computationally demanding, approximate methods were proposed to find suboptimal answers, e.g., [3].

We are also aware of a large volume of literatures on graph partitioning with various targets, METIS [8] and Mcut [2], to name a few. All these algorithms solve the graph partitioning problem with disparate objective functions, which are different from the cost model presented in this paper.

9. CONCLUSION

We study the problem of graph similarity search with edit distance constraints. Unlike the existing solutions that adopt fixed-size overlapping features for filtering, we propose a framework utilizing a novel filtering scheme based on variable-size non-overlapping partitions of data graphs. We devise a dynamic partitioning technique to enhance the filtering power, as well as an improved edit distance verification algorithm leveraging matching partitions. A cost-aware graph partitioning method is proposed to optimize the index. Empirical studies show the advantage of our method.

We observe that applications may have certain context-aware requirements (constraints); e.g., an atom `O` may change to `S` but not `C`. Although current filtering techniques do not miss such results, system performance may deteriorate under certain scenarios. As future work, we may improve the filtering power by taking advantages of these constraints.

Acknowledgements. X. Lin and W. Zhang were in part supported by NSFC61232006, NSFC61021004, ARC DP120104168, DP110102937 and DE120102144. C. Xiao was supported by FIRST Program, Japan and KAKENHI (23650047 and 25280039).

10. REFERENCES

[1] H. Bunke and G. Allermann. Inexact graph matching for structural pattern recognition. *Pattern Recognition Letters*, 1(4):245 – 253, 1983.

[2] C. H. Q. Ding, X. He, H. Zha, M. Gu, and H. D. Simon. A min-max cut algorithm for graph partitioning and data clustering. In *ICDM*, pages 107–114, 2001.

[3] S. Fankhauser, K. Riesen, and H. Bunke. Speeding up graph edit distance computation through fast bipartite matching. In *GbRPR*, pages 102–111, 2011.

[4] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, first edition edition, Jan. 1979.

[5] W.-S. Han, J. Lee, and J.-H. Lee. Turbo$_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *SIGMOD Conference*, pages 337–348, 2013.

[6] H. He and A. K. Singh. Closure-Tree: An index structure for graph queries. In *ICDE*, page 38, 2006.

[7] C. Jin, S. S. Bhowmick, B. Choi, and S. Zhou. PRAGUE: Towards blending practical visual subgraph query formulation and query processing. In *ICDE*, pages 222–233, 2012.

[8] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. In *ICPP (3)*, pages 113–122, 1995.

[9] A. Khan, Y. Wu, C. C. Aggarwal, and X. Yan. NeMa: Fast graph search with label similarity. *PVLDB*, 6(3):181–192, 2013.

[10] K. Riesen, S. Fankhauser, and H. Bunke. Speeding up graph edit distance computation with a bipartite heuristic. In *MLG*, 2007.

[11] A. Sanfeliu and K.-S. Fu. A distance measure between attributed relational graphs for pattern recognition. *IEEE transactions on systems, man, and cybernetics*, 13(3):353–362, 1983.

[12] H. Shang, X. Lin, Y. Zhang, J. X. Yu, and W. Wang. Connected substructure similarity search. In *SIGMOD Conference*, pages 903–914, 2010.

[13] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB*, 1(1):364–375, 2008.

[14] G. Wang, B. Wang, X. Yang, and G. Yu. Efficiently indexing large sparse graphs for similarity search. *IEEE Trans. Knowl. Data Eng.*, 24(3):440–451, march 2012.

[15] X. Wang, X. Ding, A. K. H. Tung, S. Ying, and H. Jin. An efficient graph indexing method. In *ICDE*, pages 210–221, 2012.

[16] X. Yan, P. S. Yu, and J. Han. Substructure similarity search in graph databases. In *SIGMOD Conference*, pages 766–777, 2005.

[17] Z. Zeng, A. K. H. Tung, J. Wang, J. Feng, and L. Zhou. Comparing stars: On approximating graph edit distance. *PVLDB*, 2(1):25–36, 2009.

[18] K. Zhang, J. T.-L. Wang, and D. Shasha. On the editing distance between undirected acyclic graphs and related problems. In *CPM*, pages 395–407, 1995.

[19] S. Zhang, J. Yang, and W. Jin. SAPPER: Subgraph indexing and approximate matching in large graphs. *PVLDB*, 3(1):1185–1194, 2010.

[20] X. Zhao, C. Xiao, X. Lin, Q. Liu, and W. Zhang. A partition-based approach to structure similarity search. Technical Report UNSW-CSE-TR-201327, 2013.

[21] X. Zhao, C. Xiao, X. Lin, W. Wang, and Y. Ishikawa. Efficient processing of graph similarity queries with edit distance constraints. *The VLDB Journal*, pages 1–26, 2013.

[22] G. Zhu, X. Lin, K. Zhu, W. Zhang, and J. X. Yu. TreeSpan: Efficiently computing similarity all-matching. In *SIGMOD Conference*, pages 529–540, 2012.

[23] Y. Zhu, L. Qin, J. X. Yu, and H. Cheng. Finding top-k similar graphs in graph databases. In *EDBT*, pages 456–467, 2012.