Information Systems 80 (2019) 91–106

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

A novel edge-centric approach for graph edit similarity computation

Karam Gouda [a,b,*], Mosab Hassaan [c]

[a] Faculty of Computers & Informatics, Benha University, Egypt
[b] Computer & Decision Engineering Department, Université Libre de Bruxelles, Belgium
[c] Faculty of Science, Benha University, Egypt

HIGHLIGHTS

- A Novel approach for computing the exact edit distance between labeled graphs.
- It leverages the relation between common sub-structures and graph edit distance.
- Efficient heuristics to cut off the huge search space.
- An exact method which can also be used as any-time approximation method.
- Experiments show the effectiveness on graph similarity search and classification.

ARTICLE INFO

Article history:
Received 27 February 2018
Received in revised form 12 August 2018
Accepted 24 October 2018
Available online 2 November 2018

Keywords:
Graph data
Edit distance
Exact methods
Graph similarity search

ABSTRACT

Graph similarity is an important notion with many applications. Graph edit distance is one of the most flexible graph similarity measures available. The main problem with this measure is that in practice it can only be computed for small graphs due to its exponential time complexity. This paper addresses the high complexity of graph edit distance computations. Specifically, we present `CSI_GED`, a novel edge-centric approach for computing graph edit distance through common sub-structure isomorphisms enumeration. `CSI_GED` utilizes backtracking search combined with a number of heuristics to reduce memory requirements and quickly prune away a large portion of the mapping search space. Experiments show that `CSI_GED` is highly efficient for computing graph edit distance; it outperforms the state-of-the-art methods by over three orders of magnitude. It also shows that `CSI_GED` scales the computation gracefully to larger and distant graphs on which current methods fail to run. Moreover, we evaluated `CSI_GED` as a stand-alone graph edit similarity search query method. The experiments show that `CSI_GED` is effective and scalable, and outperforms the state-of-the-art indexing-based methods by over two orders of magnitude.

1. Introduction

Powerful data structures such as graphs are currently used to represent complex entities and their relationships in many application areas. These areas include but are not limited to Pattern Recognition [1], Social Network [2], Software Engineering [3], Bio-informatics [4], Semantic Web [5], and Chem-informatics [6]. Yet, the expressive power and flexibility of the graph data model come at the cost of high computational complexity of many fundamental graph data tasks. Among these, computing the edit distance of a pair of graph objects is proved to be NP-hard problem [7]. Given two labeled graphs, their graph edit distance measures the minimum cost graph editing to be performed on one of them to get the other. A graph edit operation is usually one of vertex insertion/deletion, edge insertion/deletion or a change of a vertex'/edge's label in the graph.

Due to the rich information provided by its associated edit sequence as well as its ability to cope with any kind of graph structure and labeling scheme, graph edit distance is considered as one of the most flexible graph similarity measures available for labeled graphs. Today, graph edit similarity plays a significant role in managing graph data [7–10], and is employed in a variety of analysis tasks such as graph classification and clustering [11,12], object recognition in computer vision [1], etc.

Very little work has been presented to address the high complexity of graph edit similarity computation. Most of the existing methods adopt the best-first search paradigm A^* [13–16]. The basic idea of these methods is to find a vertex map between the two graphs which induces the minimum edit cost. To achieve this, the underlying vertex mapping space is explored much like traversing an ordered search tree, where intermediate tree nodes represent

* Corresponding author at: Faculty of Computers & Informatics, Benha University, Egypt.
E-mail addresses: karam.gouda@fci.bu.edu.eg, karam.gouda@ulb.ac.be (K. Gouda), mosab.hassaan@fsc.bu.edu.eg (M. Hassaan).

https://doi.org/10.1016/j.is.2018.10.003
0306-4379/

partial maps and leaf nodes represent complete ones. At each search state, the minimum-cost partial map is found and extended, and the cost of each extension is updated. This process is continued until the selected map is a complete one.

The main problem of A^*-based methods is that when comparing large graphs, huge number of partial maps need to be maintained. As a consequence, memory grows exponentially and searching for the minimum-cost partial map at each search state becomes very expensive. The time and space requirements crucially hamper A^*-based methods from being used with real-world applications. In practice, larger graphs are not uncommon. Consider, for example, the area of drug development. In order to study the properties of a new compound, the drug designer first asks the chemical compound database for those compounds which are within a specified edit distance from the new one. This step, called compound screening [17], helps the drug designer get an initial view of the compound at hand since similar compounds may have similar biological activities. The chemical compound database contains graphs with average number of vertices doubling at least that of the graphs which could be processed by A^*.

In this paper, we presents a novel approach for graph edit distance computation, named CSI_GED: **C**ommon **S**ub-structure **I**somorphisms based **G**raph **E**dit **D**istance, which minimizes memory requirements and scales to larger and *distant* (i.e., far apart from each other) graphs. CSI_GED uses a completely different approach to compute graph edit distance. Instead of mapping vertices and then deducing the edge edit cost as in A^*, CSI_GED considers mapping edges first and the edit cost on their end vertices comes directly as a by-product. Even though the edge mapping space seems to be relatively large, edges are allowed to match only if their composing vertices are consistent with already matched ones, called *common sub-structure isomorphism* restriction. This restriction reduces the search space to the smaller subspace of common sub-structure isomorphisms (CSIs for short).[1] We show in this paper that the space of CSIs is much smaller than the vertex-based mapping space especially when the graphs are sparse (Section 4). Moreover, computing the induced edit cost of each partial CSI becomes easy and straightforward as it is directly calculated from the derived common sub-structure. In contrast, computing that cost with A^*-based methods is rather expensive, and is done in a subsequent phase for each possible map extension.

CSI_GED utilizes *backtracking* to explore the space of CSIs. The most important benefit is that, the memory requirement is diminished since CSI_GED enumerates CSIs in a depth first manner, which is efficient in memory consumption. To scale backtracking search in huge CSIs spaces, CSI_GED incorporates three efficient heuristics to prune much of unpromising CSIs. These heuristics are developed based on the fact that the edit costs induced by enumerated CSIs are valid upper bounds on graph edit distance. Thus, the main goal of these heuristics is to enforce *dead search states*, i.e., nodes in the backtracking search tree accompanied by edit costs exceeding the minimum of seen upper bound values, to be encountered early in the search; thus cutting a large space from consideration. To achieve this objective, the first heuristic arranges the search space in such away that enabling fast finding of tighter upper bounds (Section 5.1), whereas the second heuristic maximizes the edit cost initially assigned to each CSI, by setting it to a global lower bound on graph edit distance (Section 5.2). The third is a lookahead heuristic, enabling the prediction of edit costs some levels ahead in the search (Section 5.3).

Experiments show that CSI_GED is highly efficient for computing graph edit distance; it outperforms the state-of-the-art A^*-based methods by over three orders of magnitude. It also shows that CSI_GED scales the computation gracefully to large and distant graphs on which A^*-based methods fail to run. Moreover, we evaluated CSI_GED as a stand-alone graph edit similarity search query method. The experiments show that CSI_GED is effective and scalable, and outperforms the state-of-the-art indexing-based methods by over two orders of magnitude.

A preliminary version of this paper appeared in [18]. This paper extends the work presented there by first providing proofs of the correctness of CSI_GED (Theorem 2, Lemma 1 and Theorem 5). Second, the lookahead pruning is improved by using *edge label indexing*, where well designed look-up tables are used to leverage the edge label information while comparing corresponding vertices. Extra experimentations are also conducted to further assessCSI_GED. These experiments are performed in order to:

- Evaluate the effect of heuristics on reducing the search space of CSI_GED.
- Evaluate how much improvements can edge label indexing bring to the lookahead pruning.
- Evaluate the quality of the first upper bound generated by CSI_GED against the state-of-the-art upper bound computation methods. The evaluation is done w.r.t. (1) approximation quality and computation time, and (2) graph classification accuracy. Results show that CSI_GED initially provides tighter upper bounds at very good response time compared to the current overestimation methods. Moreover, the accuracy of graph classification systems achieved under this first approximation is relatively high.

The remainder of this paper is organized as follows. Related work is discussed in Section 2. The problem of graph edit similarity computation and state-of-the-art computation methods are presented in Section 3. Section 4 presents the framework of CSI_GED and the motivation behind its construction. The different heuristics used to optimize CSI_GED are presented in Section 5. An application of CSI_GED to the graph edit similarity search problem appears in Section 6. The experimental results are reported in Section 7. Section 8 concludes the paper.

2. Related work

Graph edit distance was firstly connected with maximum common subgraphs by Bunk [19]. Recently, Brun et al. [20] uncovered the relation between graph edit paths and common sub-structures. They investigated under which conditions on the different costs of elementary edit operations, an optimal edit path is related to a maximum common sub-structure. Zheng et al. [21] exploited this relationship to derive lower and upper bounds of graph edit distance in the uncertain graph context, and when the edit operations are of unit costs. Different from [20] and [21], in this work, we re-discovered the same relationship and used it for developing CSI_GED: An efficient graph edit distance computation algorithm. To the best of our knowledge, we are the first to develop an approach for graph edit distance computation based on this relationship.

The heart of CSI_GED is to enumerate CSIs which quickly lead to optimal edit paths. Although many existing algorithms could be used to enumerate CSIs [6,22,23], they are all vertex-based methods and adapting them would lead to algorithms suffering from similar problems as A^*-based methods. Moreover, they could not be equipped with efficient space-pruning tools. In contrast, CSI_GED is an edge-centric backtracking search, which leverages the edit cost and accommodates efficient space-pruning heuristics.

In the past, few improvements of A^* have been considered, especially when it is used as a verifier in the filter-and-verify approach of the graph edit similarity search problem [10,16,24]. In [24] A^* is improved by starting its computations from an indexed

[1] The space of CSIs is efficiently identified by exploring edges instead of vertices, a property which will be made clear in the following sections.

common subgraph isomorphism. And, any extended partial map is stored only if its induced edit cost is within a specified edit distance, which minimizes memory. The main drawback of this improvement is when there exist more than one common occurrence of the matching subgraph. In this case A^* must run starting from every occurrence and it is not easy to share the computations among different runs. In [10,16] another improvement of A^* is introduced. It is based on previously indexed, path-based q-grams. In this improvement, A^* does not initiate the search from the matching parts (matching q-grams) as in the previous improvement, it instead starts with the mismatching ones, because these q-grams incur some edit operations which help in terminating A^* very quickly on the false positive candidate graphs. The mismatching q-grams are also used to enhance the search order, where the vertices contained by at least one of these q-grams are put before the others, and the first vertex is the one with the most infrequent label. In the interest of connectivity, ties are broken by mapping vertices in the order of a spanning tree. Such order leverages the graph connectivity which allows to quickly find edge edits. All these improvements and much more are implicitly considered by our approach.

Recently, the vertex-based approaches have witnessed new developments [25–28]. The earlier of these developments have tackled the memory overloading problem of A^*-based methods by carrying out depth-first instead of best-first search of the vertex-mapping space. Example methods are DF_GED [25] and its recent speed-up DF_GED^u [28]. DF_GED and CSI_GED are similar in that they both adopt the depth-first search paradigm and prune the search space based on the minimum editorial cost among all visited full mappings. However, they disagree on the way to achieve this; while DF_GED computes an expensive lower bound at each search state, which is very expensive with backtracking, CSI_GED employs efficient ordering and fast look-ahead heuristics which greatly minimize the size of the backtracking tree.

On the other hand, the more recent developments are tackling both the memory and efficiency issues at the same time by considering new efficient pruning heuristics of the vertex-mapping space. BSS_GED [27], A^{*+} [26] and DFS^+ [26] are new major methods. BSS_GED reduces the vertex mapping space by not enumerating the invalid and redundant vertex mappings, which are very huge in number. Moreover, new heuristics are developed to efficiently generate tighter upper and lower bounds for intermediate search states to further confine the search space. Finally, the beam-stack search paradigm is adopted, which allows for a flexible tradeoff between available memory and the time overhead of backtracking in the vertex mapping space. In [26] a unified framework is developed that can be instantiated into either a best-first search approach A^{*+} or a depth-first search approach DFS^+, and allows to use any previously developed lower bounding technique for intermediate states. In addition, anchor-aware lower bounding techniques are developed for intermediate search states, which significantly reduce the search spaces of both A^{*+} and DFS^+. Finally, to overcome the memory overloading problem of A^{*+}, intermediate search states are stored in a constant main memory space and upper bounding techniques are used to prune irrelevant states. Different from these methods, CSI_GED is a novel edge-centric mapping method based on CSIs, which are much less in number than vertex mappings. In addition, CSI_GED utilizes an ordering of the edge mapping space to facilitate fast finding of tighter upper bounds, and fast look-ahead instead of the expensive lower bounding techniques to prune the search space.

Many fast algorithms seeking suboptimal solutions to graph edit distance have been proposed. Some of these algorithms produce upper bound estimation [7,29–32]. Riesen and Bunke [29] have developed a graph mapping method, which first constructs a cost matrix between the vertices of the two graphs, and then uses a cubic-time bipartite assignment algorithm, called Hungarian algorithm [33], to optimally match the vertices. A cost matrix with an entry for each pair of vertices holds the matching edit costs between the neighborhoods of the corresponding vertices. The idea behind this heuristic is that matching vertices with similar neighborhoods induces low-cost graph mapping. A similar idea is used in [29] and [34], where the notion of vertex star (branch) is developed to capture information about the vertex neighboring vertices and edges. Recently, Gouda et al. [31,32] have developed new upper bounding technique, based on the breadth-first hierarchical views of the graphs. This technique allows to explore a quadratic space of high-quality vertex mappings. It also incorporates fast view selection methods to run using only the most effective view-pairs. Our method for ordering the search space and obtaining an initial upper bound differs from these methods in two aspects. The first is that we use a novel notion of star structure called *edge star*. Edge star captures wider local structure than vertex star. The second aspect is that we do not order and match edges based only on edge stars' cost, instead an edit cost on the star's remaining graph is added in order to capture the global as well as the local structure.

3. Preliminaries

3.1. Problem statement

Let Σ be a set of discrete-valued labels. A labeled, undirected graph G is a triple $G = (V, E, l)$, where $V = \{v_1, v_2, \ldots, v_{|V|}\}$ is a set of vertices, $E = \{e_1, \ldots, e_{|E|}\} \subset V \times V$ is a set of undirected edges, and l is a labeling function $l: V \cup E \rightarrow \Sigma$, assigning for each vertex $v \in V$ or edge $e \in E$ an alphabet character $l(v) \in \Sigma$ or $l(e) \in \Sigma$. $|V|$ and $|E|$ are called the *order* and *size* of G, resp. Let L_V and L_E denote the multi-sets of labels assigned to the vertices and edges of G, resp. L_V and L_E are multi-sets because the labels are allowed to appear more than once on the vertices and edges of the graph G. In what follows a labeled, undirected graph G is simply called a graph G unless stated otherwise, and an unlabeled version of G, i.e., its structure, is referred to as $S(G)$.

A graph $G = (V, E, l)$ is a *subgraph* of another graph $G' = (V', E', l')$ (or G' is a *supergraph* of G), denoted $G \subseteq G'$, if there exists a *subgraph isomorphism* from G to G'.

Definition 1 (*(Sub-)graph Isomorphism*)**.** A subgraph isomorphism from G to G' is an *injective* function $f : V \rightarrow V'$, such that (1) $\forall u \in V, l(u) = l'(f(u))$. (2) $\forall (u, v) \in E, (f(u), f(v)) \in E'$, and $l((u, v)) = l'((f(u), f(v)))$. If $G \subseteq G'$ and $G' \subseteq G$, G and G' are graph isomorphic to each other, denoted as $G \cong G'$.

Definition 2 (*(Maximum) Common Sub-Structure (MCS)*)**.** Given two graphs G_1 and G_2. An unlabeled graph $G = (V, E)$ is said to be a common sub-structure of G_1 and G_2 if $\exists H_1 \subseteq G_1$ and $H_2 \subseteq G_2$ such that $G \cong S(H_1) \cong S(H_2)$. A common sub-structure G is a maximum common edge (resp. vertex) sub-structure if there exists no other common sub-structure $G' = (V', E')$ such that $|E'| > |E|$ (resp. $|V'| > |V|$).

Based on Definition 2 a common sub-structure is called a *common sub-graph* if the corresponding vertices and edges agree on labels, i.e., $H_1 \cong H_2$. The maximum edge (resp. vertex) sub-graph can be defined analogously.

A graph G can be transformed into another graph by elementary edit operations consisting of inserting or deleting a vertex or an edge, or changing a vertex or edge label. Notice that a vertex can be deleted only if its incident edges have been previously deleted. Each elementary edit operation p is associated with a application-dependent cost $c(p)$, measuring the strength of the corresponding

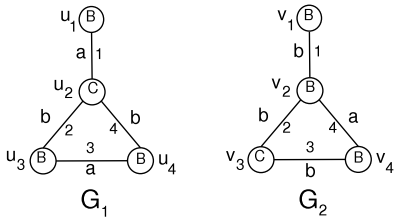

Fig. 1. Example of graphs G_1 and G_2. Numbers on edges are their ids, and each edge e_k is defined as: $e_k = (u_i, u_j)$ or $e_k = (v_i, v_j)$, $i < j$.

operation. Given two graphs G_1 and G_2, the sequence of edit operations performed on one of them to get the other is called an *edit path*. Formally, let p_i be an edit operation, an edit path $P = \langle p_i \rangle_{i=1}^{k}$ is a sequence of edit operations $\langle p_1, p_2, \ldots, p_k \rangle$ that transform G_1 into G_2, that is, $P(G_1) = G_1 \rightarrow^{p_1} G^1 \rightarrow^{p_2} G^2 \ldots \rightarrow^{p_k} G^k \cong G_2$. The cost of an edit path P is the sum of its edit operation's costs, i.e., $\mathcal{C}(G_1, G_2, P) = \sum_{i=1}^{k} c(p_i)$.

Clearly, there could be multiple edit paths that turn G_1 into G_2. An optimal edit path is the one having the minimal cost. This cost defines *graph edit distance* between G_1 and G_2, denoted $GED(G_1, G_2)$. That is, $GED(G_1, G_2) = \min_P \mathcal{C}(G_1, G_2, P)$. In this paper we assume a uniform edit cost metric, in which the cost of each edit operation amounts to one, i.e., $c(p) = 1, \forall p$. According to this metric, the optimal edit path is the one with the minimum number of edit operations.

Definition 3 (*Graph Edit Similarity Computation Problem*)**.** Given two labeled, undirected graphs G_1 and G_2, and assume a uniform edit cost metric on edit operations, the problem of graph edit similarity is to compute $GED(G_1, G_2)$.

Eqs. (1) and (2) give two simple but effective lower bounds on $GED(G_1, G_2)$ to be used throughout the paper. They are well-known as *global lower bounds*. The first bound is based on the differences in size and order of the graphs, and is given by [7] as:

$$GED(G_1, G_2) \geq \|V_1| - |V_2\| + \|E_1| - |E_2\|. \quad (1)$$

The second improves the first by taking labels as well as structure information into account, and is given by [10,16] as:

$$GED(G_1, G_2) \geq \Gamma(L_{V_1}, L_{V_2}) + \Gamma(L_{E_1}, L_{E_2}), \quad (2)$$

where $\Gamma(X, Y) = max(|X|, |Y|) - |X \cap Y|$, for sets X and Y.

Example 1. Fig. 1 shows two graphs G_1 and G_2. A minimal number of two edit operations can turn G_1 into G_2: Insertion of a new edge (u_1, u_4) with label b and a deletion of the edge (u_1, u_2). Thus, $GED(G_1, G_2) = 2$. Based on Eq. (2), the value $\Gamma(L_{V_1}, L_{V_2}) + \Gamma(L_{E_1}, L_{E_2}) = [4 - (|\{C, B, B, B\} \cap \{C, B, B, B\}|)] + [4 - (|\{a, a, b, b\} \cap \{a, b, b, b\}|)] = [4 - 4] + [4 - 3] = 1$ is a global label lower bound on $GED(G_1, G_2)$. $S(G_1)$ is both edge and vertex maximum common sub-structure since it has 4 edges and 4 vertices.

Graph edit similarity computation is known to be NP-hard problem [7]. Very little work has been introduced to address this complexity. We next overview the state-of-the-art computation methods and highlight their limitations. Hereafter, the graphs G_1 and G_2 are called the source and target graphs, resp; their edges (resp. vertices) are called the source and target edges (resp. vertices).

3.2. GED computation: A* Approach

The state-of-the-art graph edit similarity computation methods are based on the search paradigm A^*, which explores all possible one-to-one vertex maps between the source and target graphs in a best-first fashion [13–16]. A^* maintains a set of partial vertex maps with their induced edit costs, and at each search state it picks-up a minimum-cost partial map to extend, where the unmatched target vertices as well as the *null vertex* – a dummy vertex with special label – are possible candidates for extension. To guide the selection process toward the most promising partial maps, the map cost is incremented by a lower bound estimate on the edit distance between the unmapped edges and vertices. This process is continued until all source vertices are matched by the selected map. The edit cost induced by the selected map in addition to that induced by the unmapped target vertices is returned as graph edit distance.

Formally, given the source and target graphs $G_1 = (V_1, E_1, l_1)$ and $G_2 = (V_2, E_2, l_2)$. Let $u_1, u_2, \ldots, u_n$ be the source vertices given in the processing order, $f(V_1) = \{f(u_1), \ldots, f(u_{i-1})\}$ be a selected partial vertex map, and $c(f) = g(f) + h(f)$ be its associated cost, where $g(f)$ stands for the edit cost induced by the matched vertices, and $h(f)$ is a lower bound on that induced by the unmatched ones. If $i - 1 = n$ then $g(f) = g(f) + c_r$ is returned as graph edit distance, where c_r is the edit cost induced by matching a null source vertex with each unmatched target vertex; Otherwise, f is extended one item at a time as the search space is traversed. For each possible extension $f(u_i) \in (V_2 \setminus f(V_1)) \cup \{v^n\}$, where v^n is a null vertex with $l_2(v^n) \notin \Sigma$, a new partial map $f(V_1) = \{f(u_1), \ldots, f(u_{i-1}), f(u_i)\}$ is constructed and $c(f)$ is updated.

Algorithm 1 updates $g(f)$. It first evaluates the cost of matching the vertices u_i and $f(u_i)$ (lines 1–2), and then on the implied edge matching (lines 3–10) as: an edge (u_j, u_i), $j < i$, connecting u_i with an already matched vertex u_j is deleted if $(f(u_j), f(u_i))$ is not a target edge, and relabeled if $(f(u_j), f(u_i))$ has a different label. For each matched vertex u_j, $j < i$, not adjacent to u_i, an edge is inserted if $(f(u_j), f(u_i))$ is a target edge. Updating $h(f)$ depends on the heuristic used; [14] computes $h(f)$ via bipartite matching whereas [16] uses Eq. (2) as a heuristic estimate.

Algorithm 1: `Update_PED`$(G_1, G_2, f(u_i), g(f))$

```
1:  if l1(ui) ≠ l2(f(ui)) then
2:      g(f)++;      /*vertex relabeling (deletion if f(ui) = v^n)*/
3:  for each uj ∈ V1, j < i do
4:      if (uj, ui) ∈ E1 ∧ (f(uj), f(ui)) ∈ E2 then
5:          if l1(uj, ui) ≠ l2(f(uj), f(ui)) then
6:              g(f)++;                    /*edge relabeling*/
7:      if (uj, ui) ∈ E1 ∧ (f(uj), f(ui)) ∉ E2 then
8:          g(f)++;                        /*edge deletion*/
9:      if (uj, ui) ∉ E1 ∧ (f(uj), f(ui)) ∈ E2 then
10:         g(f)++;                        /*edge insertion*/
11: return g(f);
```

A^*-based methods face a number of challenges. First, most of the generated partial maps cannot be discarded and have to wait until a very late stage of the search. Thus, the larger and distant the graphs are, the larger the number of partial maps that need to be maintained and processed. As a consequence, memory grows exponentially and searching for a minimum-cost partial map to extend at each search state becomes very expensive — it requires $O(\log n)$ if priority queue is used, where n is the number of current partial maps. Second, updating $c(f)$ for each partial map f is computationally expensive, and has to be done in a separate phase for each possible map extension.

Clearly, the memory and computation overhead prevent A^*-based methods from being used with real-world applications. In this paper, to address these challenges, we propose a novel approach for graph edit distance computation named `CSI_GED`. Next, we introduce the working principle of `CSI_GED`.

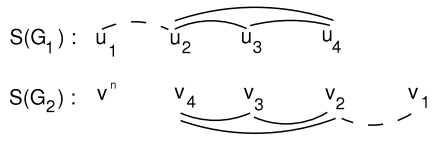

Fig. 2. The vertex map $f_3(V_1) = \{v^n, v_4, v_3, v_2\}$. Preserved edges are shown by the bold curves. Dashed curves show the unpreserved ones.

4. CSI_GED: A novel GED computation approach

In order to develop an efficient GED computation algorithm, two main concerns should be taken into account. The first is to find a way to leverage the computation of $g(f)$ for each vertex map f. The second is to develop a space traversing technique that allows searching without relying on full information of partial maps. In other words, identifying an optimal edit path must avoid the problems of A^* approach in order to scale to large and distant graphs. Below, we relate GED computation to the problem of enumerating all common sub-structures of the involved graphs.

Definition 4 (*Preserved Edges of a Vertex Map*)**.** Given two graphs $G_1 = (V_1, E_1, l_1)$ and $G_2 = (V_2, E_2, l_2)$, and a vertex map $f : V_1 \rightarrow V_2 \cup \{v^n\}$. A source edge $(u, u') \in E_1$ is preserved under f if $(f(u), f(u')) \in E_2$. A target edge $(v, v') \in E_2$ is preserved under f if there exists and edge $(u, u') \in E_1$ such that $v = f(u)$ and $v' = f(u')$.

Let $E \subseteq E_1$ and $E' \subseteq E_2$ denote the sets of preserved source and target edges under the vertex map f. The vertices in $V = \bigcup_{(u,u')\in E}\{u, u'\}$ and $V' = \bigcup_{(v,v')\in E'}\{v, v'\}$ are also preserved source and target vertices. Obviously, the graphs $G = (V, E)$ and $G' = (V', E')$ are structurally isomorphic. Thus, the graph $G = (V, E)$ composed of f's unlabeled preserved edges E and their associated unlabeled vertices V is a *common sub-structure* of G_1 and G_2 generated by f. This common sub-structure can be disconnected and is not unique in the sense that different maps can determine it.

Example 2. Consider the graphs G_1 and G_2 in Fig. 1. Define three maps $f_1, f_2, f_3 : V_1 \rightarrow V_2 \cup \{v^n\}$ as: $f_1(V_1) = \{v_1, v_2, v_3, v_4\}$, $f_2(V_1) = \{v_1, v_3, v_4, v_2\}$ and $f_3(V_1) = \{v^n, v_4, v_3, v_2\}$. Fig. 2 shows f_3. The common sub-structure generated by f_3 is given as: $G''' = (\{u_2, u_3, u_4\}, \{(u_2, u_3)(u_2, u_4), (u_3, u_4)\})$, and those generated by f_1 and f_2 are also given as: $G' = S(G_1)$ and $G'' = G'''$, resp. G' is both edge and node maximum since it has 4 edges and 4 vertices.

Theorem 1 formulates $g(f)$ of a vertex map f in terms of its generated common sub-structure, and the unpreserved edges and vertices of G_1 and G_2.

Theorem 1. *Given two graphs $G_1 = (V_1, E_1, l_1)$ and $G_2 = (V_2, E_2, l_2)$, and a vertex map $f : V_1 \rightarrow V_2 \cup \{v^n\}$. Let $G = (V, E)$ be the common sub-structure generated by f, and $G^{l_1} \subseteq G_1$ and $G^{l_2} \subseteq G_2$ be the G's corresponding subgraphs of G_1 and G_2 obtained after recovering labels. The edit cost $g(f)$ is given in terms of G as:*

$$g(f) = c_f(G^{l_1}, G^{l_2}) + |V_2 \setminus f(V_1)| + \lambda + \sum_{i=1}^{2}(|E_i| - |E|), \qquad (3)$$

where $c_f(G^{l_1}, G^{l_2})$ is the common sub-structure edit cost, $V_2 \setminus f(V_1)$ is the set of unmatched target vertices, and $\lambda = \Gamma(L_{(V_1 \setminus V)}, L_{(f(V_1)\setminus V)})$.

Proof. See [18]. ■

Based on Theorem 1, the edit cost $g(f)$ of a vertex map f can be easily computed once the common sub-structure generated by f is identified.

Example 3. Consider the vertex maps f_1, f_2 and f_3 defined in Example 2. f_3 induces 8 edit operations; it can be given in terms of its generated common sub-structure as: a deletion of the unpreserved edge (u_1, u_2), a relabeling of the unpreserved vertex u_1 (equivalent to u_1 deletion), four relabeling operations on the common sub-structure (two on the vertices u_2 and u_3, and two on the edges (u_2, u_4) and (u_3, u_4)), a vertex insertion to correspond to the unmatched target vertex v_1, and an edge insertion to correspond to the unpreserved target edge (v_1, v_2). Similarly, $g(f_1) = 5$ and $g(f_2) = 2$. Thus, the edit path induced by f_2 is the optimal one.

Example 3 shows that (1) a non-maximum common sub-structure can induce an edit cost less than that of a maximum one; (2) a common sub-structure can induce different edit costs, which are based on its generating maps.

Algorithm 2: `CSI_GED`(G_1, G_2)

```
1: Enumerate all CSIs of G1 and G2;
2: for each CSI f do
3:    compute g(f) as in Eq. (3);
4:    keep track of minimum g(f);
5: output the minimum g(f);
```

As such, a novel approach to graph edit similarity computation can be proposed. This approach suggests to enumerate all common sub-structure isomorphisms (CSIs for short) of G_1 and G_2, and calculate for each CSI its induced edit cost as in Eq. (3). The minimum of these costs is then returned as graph edit distance. This approach is named `CSI_GED` (which stands for the bold letters in: Common Sub-structure Isomorphisms based Graph Edit Distance) and outlined in Algorithm 2. Theorem 2 shows the completeness of `CSI_GED`.

Theorem 2. *Given two graphs G_1 and G_2. `CSI_GED`(G_1, G_2) returns the edit distance between G_1 and G_2.*

Proof. We prove by showing that enumerating all CSIs between G_1 and G_2 is *sufficient* and *necessary* to get their graph edit distance.
Sufficiency: Every vertex map between G_1 and G_2 defines a CSI and induces an edit path which can be determined in the context of the CSI's associated common sub-structure (Theorem 1).
Necessity: Given any edit path P, the maximum common subgraph isomorphism between G and G_2 defines a CSI between G_1 and G_2, where G is the graph obtained from G_1 after applying the deletion and relabeling operations appearing in P. ■

`CSI_GED` makes it possible to cut the overhead of computing $g(f)$ of each vertex map f. Unfortunately, similar computations are needed by any vertex-based mapping method enumerating CSIs. In such methods [6,22,23] a target vertex matches a source vertex if their connections with the previously matched vertices are consistent. To check consistency, computations similar to that of calculating the implied edge edits (Algorithm 1) are required. Meeting this challenge, `CSI_GED` constructs CSIs through mapping edges instead of vertices. We next show that mapping edges eases the consistency checking problem. We also show that the space of CSIs is considerably smaller than the full edge mapping space, and smaller than the vertex-based mapping space on real data graphs.

4.1. CSIs enumeration

Initially, to match edges, any target edge is considered as an ordered-pair of vertices. Hence, both the target edge $e = (v, v') \in E_2$ and its reverse $e^r = (v', v)$ are possible matching candidates. Let $\tilde{E}_2 = \{e, e^r : e \in E_2\}$ be an extended set of target edges. We

say that a target edge $e' = (v, v') \in \tilde{E_2}$ matches a source edge $e = (u, u') \in E_1$, denoted $e \rightarrow e'$, iff v and v' are matched with u and u', resp. Lemma 1 gives the properties that any edge map must satisfy when it comes to identify a common sub-structure.

Lemma 1. *Given two graphs $G_1 = (V_1, E_1, l_1)$ and $G_2 = (V_2, E_2, l_2)$. The map $f : E_1 \rightarrow \tilde{E_2} \cup \{e^n\}$, where e^n is a null edge with $l_2(e^n) \notin \Sigma$, is an edge map iff $e \rightarrow f(e), \forall e \in E_1$. The edge map f defines a common sub-structure if (1) it only allows many source edges to be mapped to e^n; and (2) for any two adjacent source edges $e = (u, u')$ and $e' = (u, w)$, if $f(e) \neq e^n$ and $f(e') \neq e^n$ then they must be consistent on matching the connecting vertex u.*

Proof. First, since a common sub-structure does not have to contain any particular edge, we can map one or more edges to the null edge (condition 1). Condition 2 ensures that structural connectivity must be preserved by the map. ■

Lemma 1 shows that the space of CSIs is much smaller than the original space. In what follows, each CSI is represented as a multiset of indexed edges $f(E_1) = \{e_{i_1}, \ldots, e_{i_{|E_1|}}\}$, where each e_{i_j} – the matching edge of $e_j \in E_1$ – is chosen from a finite *possible set* $P_j \subseteq \tilde{E_2} \cup \{e^n\}$, and e^n is the only edge that can be repeated in $f(E_1)$.

`CSI_GED` enumerates CSIs while traversing the edge mapping space using *backtracking*. Backtracking views edge maps as arranged in a tree-like structure, where each tree node corresponds to a partial edge map of length equal to the depth of that node. Backtracking starts with an empty map $f_0 = \{\}$ and extends it one edge at a time as the search space is traversed. Given a partial edge map $f_l = \{e_{i_0}, e_{i_1}, \ldots, e_{i_{l-1}}\}$ of length l, the possible values for the next extension e_{i_l} comes from a set $C_l \subseteq P_l$ called the *combine set*. If $e' \in P_l - C_l$, then nodes in the subtree with root node $f_{l+1} = \{e_{i_0}, e_{i_1}, \ldots, e_{i_{l-1}}, e'\}$ will not be considered by the backtracking algorithm. Since such subtrees have been pruned away from the original search space, the determination of C_l is also called *pruning*. Algorithm 3 outlines the method. The main loop in `CSI-backtrack` tries to extend f_l with every edge e' in the current combine set C_l. Line 5 computes f_{l+1}, which is simply f_l extended with e'. e' and its reverse e'^r are then marked as matched at Line 6. In Line 8, the new possible set of extensions P_{l+1} is extracted which consists only of target edges $e \in \tilde{E_2}$ that are not matched yet. A new combine set consisting of all valid extensions is then created for the next pass at Line 9. A target edge is a *valid* extension if its end vertices are conforming with the previously matched ones (Lemma 1).[2] Thus, the combine set C_{l+1} comprises those edges in P_{l+1} that produce a common sub-structure when used to extend f_{l+1}. Any edge not in the combine set refers to a pruned subtree. Line 10 recursively calls `CSI-backtrack` for each extension. At Line 11, when all source edges are matched, the map is added to $\mathcal{CSI}$ – the set of all CSIs. Theorem 3 gives an upper bound estimation on the size of CSIs space.

Theorem 3. *The space of CSIs is of size $O(|E_2| \times (\frac{|V_2|}{2} - 2)! \times (d-1)^{|E_1| - \frac{|V_1|}{2}})$, where $d = max_{u \in V_2}(deg(u))$ – the maximum vertex degree in the target graph.*

Proof. See [18]. ■

Theorem 3 shows that the space of CSIs is much smaller than $O(|V_2|^{|V_1|})$ – the size of vertex-based mapping space – especially when the graphs are sparse,[3] because in sparse graphs $|E_1|$ is very close to $|V_1|$, $d \ll |V_2|$, and $(\frac{|V_2|}{2} - 2)! \ll (\frac{|V_2|}{2} - 2)^{\frac{|V_1|}{2}}$. Only when the target graph is complete, i.e., $d = |V_2| - 1$, the vertex and edge spaces have almost the same size. In cases where both graphs are very dense, i.e., $|E_1| \gg |V_1|$ and d approaching $|V_2|$, the vertex mapping space becomes smaller.

Algorithm 3: `CSIs_Enum`(G_1, G_2)

```
1  1: CSI = ∅;                                   /*a set to hold all CSIs*/
2  2: CSI-backtrack(∅, Ẽ2 ∪ {e^n}, 0);
3  3: return CSI;
   CSI-backtrack(f_l, C_l, l)
   4:  for each e' ∈ C_l
   5:    f_{l+1} = f_l ∪ {e'};
   6:    if e' ≠ e^n then mark e' and e'^r as matched;
   7:    if l < |E_1| − 1 then
   8:      P_{l+1} = {e : e ∈ Ẽ2 and e is unmatched};
   9:      C_{l+1} = CSI-combine(f_{l+1}, P_{l+1});
   10:     CSI-backtrack(f_{l+1}, C_{l+1}, l + 1);
   11:   else CSI = CSI ∪ {f};      /*f is a complete CSI*/
   12:   f_{l+1} = f_{l+1} \ {e'};          /*restore state(lines 12–13)*/
   13:   if e' ≠ e^n then mark e' and e'^r as unmatched;
   //Can f_{l+1} combine with edges in P_{l+1}?
   CSI-combine(f_{l+1}, P_{l+1})
   14:  C = ∅;
   15:  for each e ∈ P_{l+1}
   16:    if e is a valid extension then C = C ∪ {e};
   17:  return C ∪ {e^n};
```

Example 4. Fig. 3 shows part of the full edge mapping search tree of the graphs G_1 and G_2 given in Fig. 1, where nodes at level i indicate the target edges possible for matching the source edge $e_i \in E_1$. As shown, a source edge matches a target edge at a time according to the given order of source edges. So, inner tree nodes correspond to partial edge maps and leave nodes correspond to complete ones. Backtracking search space can be considerably smaller than the full space. For example, it starts with $f_0 = \{\}$ and $C_0 = \tilde{E_2} \cup \{e^n\}$. At level 1, each item in C_0 is added to f_0 in turn. For example, $e_1 = (v_1, v_2)$ is added to obtain $f_1 = \{e_1\}$. Then e_1 and e_1^r are marked as matched. The possible set P_1 of e_1 comprises all target edges in $\tilde{E_2}$ that are not matched yet. However, since $e_2 = (v_2, v_3)$, $e_4 = (v_2, v_4)$ and e^n are the only valid extensions, the subtrees rooted at e_2^r, e_3, e_3^r and e_4^r are pruned. Considering this common sub-structure isomorphism restriction, the search space rooted at $f = \{e_1\}$ is reduced from 139 edge maps to only 13 CSIs to examine for edit distance. The whole search space of this example contains 51 CSIs in total to examine. Contrast this with $|V_2|^{|V_1|+1} = 4^5 = 1024$ vertex maps that need to be examined in A^*-based methods. There is around 95% reduction of the search space.

Unfortunately, the number of CSIs becomes quite large with large graphs. Large search space is a major challenge for backtracking. To meet this challenge and scale `CSI_GED`, three heuristics, described below, are developed to cut off the search space.

5. Optimizing CSI_GED

Since CSIs are enumerated in a depth first manner, some CSIs will be available early in the search before others. The edit costs induced by the enumerated ones are in fact upper bounds of graph edit distance, and can be used to cut branches of the backtracking tree at the nodes associated with (expected) higher edit costs. To facilitate this pruning, instead of computing the edit cost $g(f)$ induced by each CSI f in a separate, subsequent phase as in Algorithm

[2] Edge validity is easily checked by maintaining a vertex map M with each edge map f, to store the matching vertices of already matched edges. When $(v, v') \in E_2$ extends f for $(u_i, u_j) \in E_1$, the nonempty slot $M(i)$ or $M(j)$ must be equal to v or v', resp.

[3] Graph density is given by $\frac{2|E|}{|V|(|V|-1)}$, i.e., the number of edges in the graph proportion to the number of edges if the graph is complete.

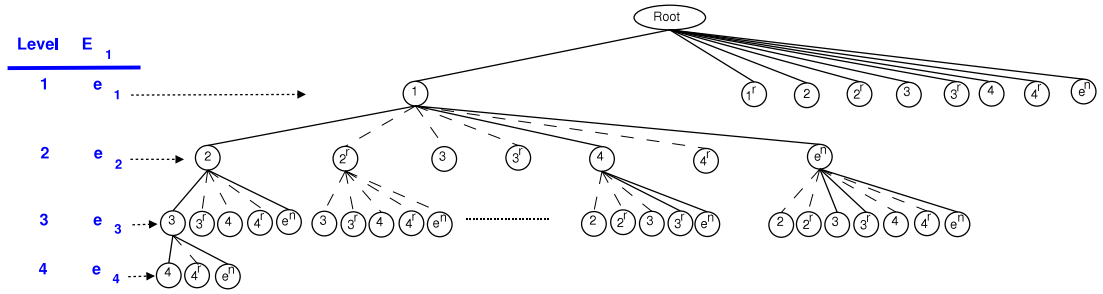

Fig. 3. Edge mapping/backtrack search tree. Black edges are considered by backtracking and dashed ones show pruned branches. e^n: null edge.

2, incremental computation of $g(f)$ must be supported. Theorem 1 shows that $g(f)$ is computed using five independent costs. The first cost, $c_f(G^{l_1}, G^{l_2})$, calculates the edits on the two subgraphs $G^{l_1} \subseteq G_1$ and $G^{l_2} \subseteq G_2$ which are identified based on the common sub-structure $G = (V, E)$ of G_1 and G_2. The second and third costs calculate the number of unpreserved source and target edges, which are $|E_1| - |E|$ and $|E_2| - |E|$, resp. The last two costs come from the edits on the source and target vertices which are not in G: $|V_2 \setminus f(V_1)|$ vertex insertions for the unmatched target vertices (the fourth cost) and the relabeling required on the unpreserved source vertices (the last cost).

Only the first two costs could be easily injected into the CSIs construction process. Given a search state, the common sub-structure identified at this state is expanded if a valid map extension is found; otherwise, it remains unchanged and the source edge is deleted by matching it with e^n. Thus, having the edit cost on the valid extension increments $g(f)$ based on the first cost value. Also, deleting the source edge increments $g(f)$ based on the second cost value. The third and fifth costs could also be injected but at extra computations (an easy injection of the third cost will be given later in this section); thus they remain to be calculated subsequently, i.e., after completing the map. Since $\|V_1|-|V_2\|$ (a fraction of the fourth value) is global and independent of any CSI, it could be used as an initial cost for each CSI. The new code of `CSI_GED` after cost injection is given by Algorithm 4. It is a straightforward extension of Algorithm 3. The main addition is the injection of $g(f)$ based on the first two cost values to eliminate branches of the backtracking tree. In addition to the main steps of Algorithm 3, the new code starts with an hypothetical upper bound $A = \infty$ and an initial cost $IC = \|V_1|-|V_2\|$ assigned to every CSI f. It adds a step to update $g(f)$ after the map extension (line 7) and a step to update A (lines 14–15). To include pruning based on the upper bound value, a new condition is added to the main pruning step at line 21. The *edge matching cost* $emc(e \to e')$ that is used for incrementing the first two costs is defined as follows.

Definition 5 (*Edge Matching Cost*)**.** Given a source and target edges $e = (u, u')$ and $e' = (v, v')$. The cost of assigning e' to e, called edge matching cost and denoted $emc(e \to e')$, is given as:

$$emc(e \to e') = \begin{cases} c(u \to v) + c(u' \to v') + c(e \to e'), & e' \neq e^n; \\ 1, & e' = e^n \end{cases}$$

where the cost function c returns 0 if the two matching items have identical labels, and 1 otherwise.[4]

It is clear that the two cases in Definition 5 update $g(f)$ on the first two cost values, resp.

Algorithm 4: `CSI_GED`(G_1, G_2)

```
1 (* indicates a new line not in Algorithm 3, and
2 Δ = λ + |V2 \ f(V1)| − ||V1|−|V2||).
1: *A = ∞;                    //initial upper bound on GED(G1, G2).
2: *IC = ||V1|−|V2||;  //initial edit cost for each CSI.
3:  CSI-backtrack(∅, Ẽ2 ∪ {e^n}, 0, IC);
4:  return A;
CSI-backtrack(fl, Cl, l, g(fl))
5:  for each e' ∈ Cl
6:    fl+1 = fl ∪ {e'};
7: *  g(fl+1) = g(fl) + emc(el+1 → e');
8:    if e' ≠ e^n then mark e' and e^r as matched;
9:    if l < |E1| − 1 then
10:     Pl+1 = {e : e ∈ Ẽ2 and e is unmatched};
11:     Cl+1 = CSI-combine(fl+1, Pl+1, g(fl+1));
12:     CSI-backtrack(fl+1, Cl+1, l + 1, g(fl+1));
13:   else /*a complete CSI*/
14:*    if g(fl+1) + |{e ∈ E2 : e is unmatched}|+Δ < A
15:*      A = g(fl+1) + |{e ∈ E2 : e is unmatched}|+Δ ;
16:   fl+1 = fl+1 \ {e'}; /*restore state (lines 16–17)*/
17:   if e' ≠ e^n then mark e' and e^r as unmatched;
18:*  g(fl+1) −= emc(el+1 → e');
CSI-combine(fl+1, Pl+1, g(fl+1))
19:  C = ∅;
20:  for each e ∈ Pl+1
21: *  if e is valid & g(fl+1) + emc(el+2 → e) < A
22:     C = C ∪ {e};
23:  return C ∪ {e^n};
```

To boost pruning based on upper bounds, a sum of $g(f)$ with a lower bound on the edit distance of unmapped edges and vertices could be used. However, it is not practical to compute a lower bound at every tree node of an ever expanding search tree. Here, new efficient pruning heuristics are developed. The first is an ordering heuristic, arranges the search space in such away that enabling fast finding of tighter upper bounds (Section 5.1), whereas the second heuristic maximizes the edit cost initially assigned to each CSI, by setting it to a global lower bound on graph edit distance (Section 5.2). The third is a lookahead heuristic, enabling the prediction of edit costs some levels ahead in the search (Section 5.3). Such heuristics will allow tree nodes accompanied with worse edit costs than the best seen upper bounds to be encountered early in the search; thus cutting many branches from consideration. Next, we detail each of these heuristics.

5.1. Ordering heuristic

Given that every valid extension of every partial CSI of size l comes from the same set of target edges $\tilde{E}_2$ (line 10, Algorithm

[4] Notice that since we are matching edges, based on Definition 5 it is possible that a target vertex be assigned more than once to a source vertex. We should take care of this in the implementation and evaluates the cost of matching vertices only once.

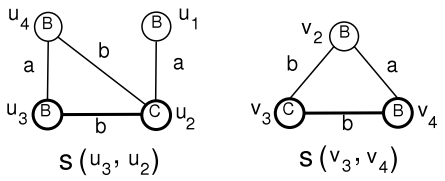

Fig. 4. Example of two edge stars $s(u_3, u_2)$ and $s(v_3, v_4)$.

4), $\tilde{E_2}$ could be ordered at each search level l in such a way that those CSIs which produce tighter upper bounds are enumerated first. The ordering heuristic we use arranges $\tilde{E_2}$ at a search level l in increasing order of $\mathcal{C}(e_l, e')$, where $\mathcal{C}$ computes an approximation of GED given that the target edge e' is assigned to the source edge e_l. In order to define $\mathcal{C}$, *edge star* – a local structure around an edge – and *star matching cost* will be first defined.

Definition 6 (*Edge Star*)**.** Given a graph G and an edge $e \in G$. The edge star of e, denoted $s(e)$, is a subgraph consisting of e, called the star core, and edges adjacent to e.

Definition 7 (*Star Matching Cost*)**.** Given two source and target edges $e = (u, u')$ and $e' = (v, v')$. The star matching cost of $s(e)$ and $s(e')$, denoted by $smc(e, e')$, is given as:

$$smc(e, e') = emc(e \rightarrow e') + \Gamma(LE_u, LE_v) + \Gamma(LE_{u'}, LE_{v'}),$$

where LE_x is the set of labels of edges incident on the vertex x, excluding the core's label.

Note that the outer vertices of each edge star are not involved in computing the star matching cost.

Definition 8 (*Star Remaining Graph*)**.** Given a graph G and an edge star $s(e)$ of $e \in G$. The star remaining graph of $s(e)$, denoted G^e, is defined to be the graph obtained after removing $s(e)$ from G.

Given two source and target edges e_l and e'. Let $G_1^{e_l} = (V_1', E_1', l_1)$ and $G_2^{e'} = (V_2', E_2', l_2)$ be the star remaining graphs of $s(e_l)$ and $s(e')$, resp. The cost function $\mathcal{C}(e_l, e')$ is defined to be the star matching cost of $s(e_l)$ and $s(e')$ in addition to the global lower bound on the edit distance between the stars' remaining graphs, i.e., $\mathcal{C}(e_l, e') = smc(e_l, e') + \Gamma(L_{V_1'}, L_{V_2'}) + \Gamma(L_{E_1'}, L_{E_2'})$.

Example 5. Consider the graphs G_1 and G_2 in Fig. 1, and the source and target edges $e = (u_3, u_2)$ and $e' = (v_3, v_4)$. Fig. 4 shows the edge stars $s(e)$ and $s(e')$. The star remaining graphs of e and e' are given as: $G^e = (\{u_1, u_4\}, \emptyset, l_1)$ and $G^{e'} = (\{v_1, v_2\}, \{(v_1, v_2)\}, l_2)$. The matching cost of $s(e)$ and $s(e')$ is calculated as: $smc(e, e') = emc(e \rightarrow e') + \Gamma(LE_{u_3}, LE_{v_3}) + \Gamma(LE_{u_2}, LE_{v_4}) = 2 + 1 + 1 = 4$. The vertex and edge global bounds on GED of the remaining graphs are calculated as 0 and 1, resp. Thus, $\mathcal{C}(e, e') = 4 + 0 + 1 = 5$. If the source edges are processed as in the order $E_1 = \{(u_1, u_2), (u_2, u_3), (u_3, u_4), (u_4, u_2)\}$, then the target edges at level 2, e.g., are ordered based on $\mathcal{C}$ as: $\{(v_3, v_4), (v_3, v_2), (v_2, v_3), (v_2, v_1), (v_4, v_3), (v_2, v_4), (v_4, v_2), (v_1, v_2)\}$, where $\mathcal{C}$ is calculated for every target edge w.r.t. the source edge (u_2, u_3), and given as: $\{2, 2, 3, 4, 5, 5, 6, 6\}$.

5.2. Maximizing initialization cost

It is possible to maximize the initial cost assigned to each CSI to include graph size as well as graph order differences. That is, the initial cost could be refined to become $IC = \|V_1| - |V_2\| + \|E_1| - |E_2\|$. Adding the difference of graph sizes to the initial cost entails modifying the edge matching cost *emc*, especially on the deleted source edges. That is, in the process of matching edges, instead of assessing edge deletion as of edit cost one (Definition 5), it is modified taking into account the different values of $|E_1|$ and $|E_2|$. Theorem 4 defines the new edge deletion cost.

Theorem 4. *Given an empty CSI f whose initial cost $g(f) = \|V_1| - |V_2\| + \|E_1| - |E_2\|$, the cost of deleting a source edge e while extending f is given as:*

$$emc(e \rightarrow e^n) = \begin{cases} 2, & |E_1| \leq |E_2|; \\ 2, & |E_1| > |E_2| \,\&\, k \geq (|E_1| - |E_2|); \\ 0, & |E_1| > |E_2| \,\&\, k < (|E_1| - |E_2|), \end{cases}$$

where k is the number of previously deleted source edges.

Proof. See [18] ■

In addition to maximizing the initial cost, Theorem 4 allows smooth injection of the edit cost of unpreserved target edges, $|E_2| - |E|$, into the CSIs enumeration process. Algorithm 4 is modified accordingly in order to accommodate the new initialization and edge deletion cost. The number of unpreserved target edges is also removed from the subsequent calculations at lines 14 and 15.

5.3. Look-ahead based pruning

Consider the graphs G_1 and G_2 given in Fig. 5. These two graphs have the same order, but G_1 has one extra edge. Thus, the edit cost assigned to each CSI is initialized to one. Consider a partial CSI f that matches the bold edges of both graphs. Its edit cost $g(f)$ remains equal to one because the subgraphs associated with this common sub-structure induce no cost. If the current best upper bound is greater than one, it is not possible to stop extending f at this stage based on $g(f)$ alone. Here, we introduce another cost function $g'(f)$ effective for pruning such cases, to be maintained with each CSI f in addition to $g(f)$. This new function implements *lookahead*. That is, it is able to calculate the edit cost some levels ahead in the search tree.

Given a graph G, define for each subgraph $H \subseteq G$ two neighborhood structures, called *inner* and *outer neighborhoods* as follows.

Definition 9 (*Inner & Outer Neighborhoods of a Subgraph*)**.** Given a graph G. Let H be a subgraph of G. The inner neighborhood of H, denoted $N_I(H)$, is defined as: $N_I(H) = \{(u, v) \in G : u, v \in H\}$. The outer neighborhood of H, denoted $N_O(H)$, is defined as: $N_O(H) = \{(u, v) \in G : u \in H \wedge v \notin H\}$.

Based on the inner and outer neighborhoods of a subgraph $H \subseteq G$, the *inner* and *outer neighborhood* of a vertex $u \in H$ is defined as: $N_I(u) = \{(u, v) \in N_I(H)\}$ and $N_O(u) = \{(u, v) \in N_O(H)\}$, resp. $|N_I(u)|$ and $|N_O(u)|$ define the *inner* and *outer degree* of $u \in H$ denoted as $d_I(u)$ and $d_O(u)$, resp.

Now, given two graphs G_1 and G_2, and a (partial) CSI f. Let M be the vertex map associated with f, and $G^{l_1} \subseteq G_1$ and $G^{l_2} \subseteq G_2$ be the subgraphs identified based on f's common sub-structure. The cost $g'(f)$ is defined, in terms of the inner and outer neighborhoods of G^{l_1} and G^{l_2}, as the total of four costs: the degree cost c_d on corresponding vertices, the edge cost c_e on corresponding inner edges, the cost c_r on the remaining edges, and the cost κ on vertex relabeling of G_1 and G_2 taken by the CSI f. That is, $g'(f) = c_d + c_e + c_r + \kappa$, where the degree cost c_d is given as:

$$c_d = \sum_{u \in G^{l_1}} |d_O(u) - d_O(M(u))| + \frac{1}{2}|d_I(u) - d_I(M(u))|, \tag{4}$$

where the fraction $\frac{1}{2}$ is introduced because each inner edge is used twice in the degree calculation, one for each end vertex. The inner edge cost c_e is given as:

$$c_e = \sum_{u,u' \in G^{l_1}} c((u, u') \rightarrow (M(u), M(u'))), \tag{5}$$

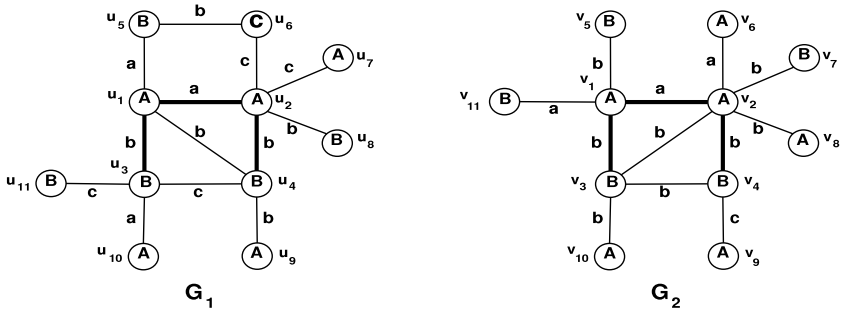

Fig. 5. Example of two graphs G_1 and G_2.

where $(u, u') \in N_l(G^{l_1})$ and $(M(u), M(u')) \in N_l(G^{l_2})$, and the cost function c returns 0 if the mapping edges have identical labels, and 1 otherwise. The remaining edge cost, c_r, is given as:

$$c_r = |n_1 - n_2|, \tag{6}$$

where $n_i = |E_i \setminus (N_I(G^{l_i}) \cup N_O(G^{l_i}))|$, $i = 1, 2$. The vertex relabeling cost, κ, is given as:

$$\kappa = max(\Gamma(L_{V_1}, L_{V_2}), h + \|V_1|-|V_2\|), \tag{7}$$

where h is the number of vertex relabeling on matched vertices.

Theorem 5. *Given two graphs G_1 and G_2, and a complete CSI f. For any partial CSI f' of f, $g'(f') \leq g(f)$.*

Proof. The partial map f' constitutes a common structure which is a substructure of the one identified by f. The edit cost on matching edges of the smaller structure are taken care by Eq. (5). Eq. (5) also evaluates the cost on inner edges that are sure to match at next tree levels, and will be part of the bigger structure. Source inner and outer edges that are to be deleted in the future because the counterpart target edges are not exist, are considered by inner and outer degree differences (Eq. (4)), resp. The inner and outer degree differences also consider the edge additions that will be taken place at the source graph because of the existence of counterpart target edges. Edges that are neither inner nor outer in both graphs and need to be added to or to be deleted from the source graph because of the missing counterparts, are taken care by the edge difference in Eq. (6). Finally, since $\Gamma(L_{V_1}, L_{V_2})$ and $\|V_1|-|V_2\|$ are both global lower bounds on vertex relabeling of G_1 and G_2, the vertex relabeling taken by a complete CSI f is at least any of these values. Because the h vertex relabeling performed by the partial map f' are done on vertices other than those in the difference $\|V_1|-|V_2\|$, h is added to $\|V_1|-|V_2\|$. Thus, the number of vertex relabeling of f is at least κ (Eq. (7)). ■

Example 6. Consider G_1 and G_2 in Fig. 5. Let f be the partial CSI that matches the bold edges in both graphs, $M = \{v_1, v_2, v_3, v_4\}$ be the vertex map associated with f and G be the common substructure identified by f. The inner neighborhoods of G^{l_1} and G^{l_2} are given as: $N_I(G^{l_1}) = \{(u_1, u_2), (u_1, u_3), (u_1, u_4), (u_2, u_4), (u_3, u_4)\}$ and $N_I(G^{l_2}) = \{(v_1, v_2), (v_1, v_3), (v_2, v_3), (v_2, v_4), (v_3, v_4)\}$, resp. The outer neighborhoods are $N_O(G^{l_1}) = E_1 \setminus (N_I(G^{l_1}) \cup \{(u_5, u_6)\})$ and $N_O(G^{l_2}) = E_2 \setminus N_I(G^{l_2})$, resp. The inner and outer degrees of a vertex u_1, e.g., are given as: $d_I(u_1) = 3$ and $d_O(u_1) = 1$. The lookahead cost $g'(f) = 4 + 1 + 1 + 2 = 8$, where the degree cost is calculated as: $c_d = (1+\frac{1}{2})+(0+\frac{1}{2})+(1+\frac{1}{2})+(0+\frac{1}{2}) = 4$, the inner edge cost as: $c_e = 0 + 0 + 0 + 1 = 1$, the remaining edge cost as: $c_r = |1 - 0| = 1$, and the vertex relabeling cost as: $\kappa = max(11 - 9, 0 + 0) = 2$. Hence, if the current upper bound value is, e.g., 7, the subtree rooted at f could be pruned based on $g'(f)$.

Besides being very effective for pruning the search space, computing $g'(f)$ for any partial CSI f is not computationally-demanding as the costs c_d, c_e and c_r (Eqs. (4)–(6)) are easy to compute, and the costly $\Gamma(L_{V_1}, L_{V_2})$ (Eq. (7)) could be calculated once at the beginning of the algorithm and be used for every CSI.

5.3.1. Improving lookahead using label indexing

The lookahead cost g' includes in its computation the value c_d– the degree cost on corresponding vertices of the common substructures. The cost c_d can be modified to include edge label comparison instead of just simply comparing the number of incident edges (i.e., vertex degrees). Since edge label comparison is finer than degree comparison, c_d is guaranteed to be maximized; consequently g'. Consider Example 7, the outer degree difference $|d_O(u_2) - d_O(v_2)|$ on the corresponding vertices u_2 and v_2 does not contribute to g'. However, if we instead take edge label into account during comparison, g' can be increased to 10 instead of 8 as two edit operations are required to modify the labels on the outer edges of u_2 and v_2. The modification of c_d, denoted c_d^l, is given by the following equation.

$$c_d^l = \sum_{u \in G^{l_1}} \Gamma(L_{N_O(u)}, L_{N_O(M(u))}) + \frac{1}{2}|d_I(u) - d_I(M(u))|, \tag{8}$$

Eq. (8) shows that c_d is only modified based on the first term (outer edge comparison). The second term of c_d (inner edge comparison) remains intact because the label comparison of inner edges is already taken care by Eq. (4).

Though c_d^l may improve the lookahead pruning, it is computationally expensive to calculate $\Gamma(L_{N_O(u)}, L_{N_O(M(u))})$ with ever growing common sub-structures and changing of vertex's outer neighborhood. To overcome this computation challenge we construct a set of look up tables for caching the computation as follows. Given two graphs G_1 and G_2 to be compared. For each graph G_i, i =1, 2, we calculate its neighborhood sets $\{L_{N(u)}\}_{u \in G_i}$, where $N(u)$, $u \in G_i$, is the set of u's incident edges in G_i, i.e., $N(u) = \{(u, w) : (u, w) \in G_i\}$. Obviously, each of these collections initially determines the outer neighborhoods of graph vertices. The inner neighborhood of each vertex, on the other hand, is initially empty. Then, from each neighborhood collection we enumerate all distinct subsets $\{\mathcal{P}(L_{N(u)})\}_{u \in G_1}$, where $\mathcal{P}(X)$ denote the powerset of a given set X. It is clear that the outer neighborhood of each vertex u, $L_{N_O(u)}$, at any search state of `CSI_GED` will be one of those subsets. A look-up table, called Γ table, is created to maintain the Γ computations between distinct neighborhoods of the two graphs, that is, for each two corresponding neighborhoods $X = L_{N_O(u)}$ and $Y = L_{N_O(M(u))}$, the value $max(|X|, |Y|) - |X \cap Y|$ is cached. Fortunately, the Γ table has manageable size because the number of distinct subsets in each collection is not that large as many vertices in the graph share the same neighborhood and the size of each subset is bounded by the large vertex degree.

Given two corresponding vertices, how to locate their corresponding value in the Γ table. To answer this, for each graph G_i,

N.index	i_1	i_2	i_3	i_4	i_5	i_6
DNs	$\emptyset$	a	b	ab	bb	abb

(a)

N.index	j_1	j_2	j_3	j_4	j_5	j_6
DNs	$\emptyset$	a	b	ab	bb	abb

(b)

Fig. 6. The distinct neighborhood subsets (DNs) and their position indices (N.index) of (a) G_1, (b) G_2.

Γ	j_1	j_2	j_3	j_4	j_5	j_6
i_1	0	1	1	2	2	3
i_2	1	0	1	1	2	2
i_3	1	1	0	1	1	2
i_4	2	1	1	0	1	1
i_5	2	2	1	1	0	1
i_6	3	2	2	1	1	0

Fig. 7. $\Gamma(i_k, j_l)$ for each pair of neighborhood subsets at positions i_k and j_l.

Vertex.id	u_1	u_2	u_3	u_4
N.index	i_2	i_6	i_4	i_4

(a)

Vertex.id	v_1	v_2	v_3	v_4
N.index	j_3	j_6	j_5	j_4

(b)

Fig. 8. The Accessibility Bridges of (a) G_1, (b) G_2.

T_{G_1}	i_1	i_2	i_3	i_4	i_5	i_6
a	i_1	i_1	i_3	i_3	i_5	i_5
b	i_1	i_2	i_1	i_2	i_3	i_4

(a)

T_{G_2}	j_1	j_2	j_3	j_4	j_5	j_6
a	j_1	j_1	j_3	j_3	j_5	j_5
b	j_1	j_2	j_1	j_2	j_3	j_4

(b)

Fig. 9. The transition tables of indexed subsets and edge labels.

Vertex.id	u_1	u_2	u_3	u_4
N.index	i_1	i_5	i_4	i_4

(a)

Vertex.id	v_1	v_2	v_3	v_4
N.index	j_1	j_4	j_5	j_4

(b)

Fig. 10. The Accessibility Bridges of (a) G_1, (b) G_2, after matching (u_1, u_2) with (v_1, v_2).

we maintain an array of size $|V_i|$, called Accessibility Bridge (AB) array, to store for each vertex the position index of its current outer neighborhood at the set of distinct neighborhoods. Using these indices we look up the Γ table for the relevant Γ value. Since at each iteration of CSI_GED a common sub-structure is growing by one edge at a time, the outer neighborhoods $N_O(u)$ and $N_O(M(u))$ of the involved vertices is also changing by one edge label at a time. To help update the position indices of outer neighborhoods of these vertices, another lookup table, called transition table and denoted $T_G()$, is created between the distinct neighborhoods and distinct edge labels in each graph G_i. For each neighborhood set X and an edge label a, $T_G(a, X)$ preserves the position index of the set $Y = X \setminus \{a\}$, i.e., of the neighborhood set obtained after removing the label a from X.

Example 7. Consider the graphs G_1 and G_2 in Fig. 1. The neighborhood collections of both graphs are given as: $\{L_{N(u_i)}\}_{u_i \in G_1} = \{\{a\}, \{a, b, b\}, \{a, b\}, \{a, b\}\}$ and $\{L_{N(v_i)}\}_{v_i \in G_2} = \{\{b\}, \{a, b, b\}, \{b, b\}, \{a, b\}\}$. The distinct subsets of both collections are calculated as: $\{\emptyset, \{a\}, \{b\}, \{a, b\}, \{b, b\}, \{a, b, b\}\}$. Fig. 6 shows these subsets with their indices. The Γ values for these neighborhood subsets are given in Fig. 7. The bridge arrays for both graphs are given in Fig. 8. The new indices of neighborhood subsets obtained after removing distinct edge labels, i.e. transitions tables, are given in Fig. 9. Finally, Fig. 10 shows the new accessibility values after matching the source edge (u_1, u_2) with the target edge (v_1, v_2).

6. Application: Graph edit similarity search problem

Graph edit distance is extensively used in the solution of graph edit similarity search (GESS) problem. Given a set of data graphs $\mathcal{D} = \{G_1, G_2, \ldots, G_{|\mathcal{D}|}\}$, the GESS problem is to retrieve data graphs that are similar to a given query graph Q within an edit threshold τ, that is, retrieve G_i if $GED(Q, G_i) \leq \tau$. The well-known approach to this problem, called filter-and-verify, is to first filter unpromising data graphs based on lower bounds of GED and then verify the remaining ones using the expensive edit distance computations [7,9,10,16,24,34,35]. Upper bounds of GED could also be used to exempt some valid candidates from the expensive verification phase [7]. CSI_GED can benefit GESS problem by two ways: (1) as a verifier with any GESS filtering method[5]; (2) as a stand-alone GESS query method.

Incorporating τ into the computation can further optimize CSI_GED as follows. First, the possible set P_l at each search level l could be refined by removing a target edge e' if $\mathcal{C}(e_l, e')$ is greater than τ. Indeed, those edges would not be part of any optimal CSI between the query and answering graphs. Likewise, the combine set C_l can be further refined by adding a new pruning condition based on τ: A valid target edge e' is removed from the combine set C_l if, in addition to the upper bound based pruning, $g(f)+emc(e_l \rightarrow$

[5] Deploying CSI_GED as a verifier with the filter-and-verify GESS systems which utilize tighter upper-bounds for validity checking of candidates would give ultra-fast response to the GESS queries as these upper-bounds could be used to initialize CSI_GED when verifying those candidates.

e') $> \tau$, or if the lookahead cost $g'(f)$ of f after being extended based on e' is greater than τ, i.e. if $g'(f) > \tau$. Similar to the previous argument, those edges would not be part of any optimal CSI to the answering graphs. Finally, getting a complete CSI f with $g(f)$ at most τ stops the algorithm, and the data graph is reported as an answer graph. This is due to the fact that $g(f)$ is an upper bound of graph edit distance, and in this case, the data graph is surely lying within a distance τ from the query. For data graphs whose edit distance is far less than τ, halting the algorithm becomes very quick. Adding new steps to accommodate these optimizations makes CSI_GED an efficient GESS query method.

7. Experimental results

Here, we present a comprehensive experimental study on CSI_GED. All experiments were performed on a 3 GHz Dual Core CPU with 4G RAM running Linux. CSI_GED is implemented in C++ with STL library support and compiled with GNU GCC.

Benchmark Datasets: We chose several real and synthetic graph datasets for testing the performance of CSI_GED. The real graphs are known to be sparse while the synthetic ones are always dense.

(1) **AIDS** (http://dtp.nci.nih.gov/docs/aids/aidsdata.html) is a DTP AIDS Antiviral Screen chemical compound dataset. It consists of 42,687 chemical compounds, with an average of 46 vertices and 48 edges. Compounds are labeled with 63 distinct vertex labels but the majority of these labels are H, C, O and N. The total number of distinct edge labels is 3.

(2) **Linux** (www.comp.nus.edu.sg/~xiaoli10/data/segos/linux_segos.zip) is a Program Dependence Graph (PDG) dataset generated from the Linux kernel procedure. PDG is a static representation of the data flow and control dependency within a procedure. In the PDG graph, an vertex is assigned to one statement and each edge represents the dependency between two statements. PDG is widely used in software engineering for clone detection, optimization, debugging, etc. The Linux dataset has in total 47,239 graphs, with an average of 45 vertices each. The graphs are labeled with 36 distinct vertex labels, representing the roles of statements in the procedure, such as "declaration", "expression", "control-point", etc. The edges are unlabeled.

(3) **Chem_1M** is a chemical compound dataset. It is a subset of PubChem (https://pubchem.ncbi.nlm.nih.gov) and consists of one million graphs. It has 23.98 vertices and 25.76 edges on average. The number of distinct vertex and edge labels are 81 and 3, resp.

(4) **Protein** (http://www.iam.unibe.ch/fki/databases/iam-graph-database/)is a dataset from the Protein Data Bank, constituted of 600 protein structures, with an average of 32.63 vertices each. Vertices represent secondary structure elements and are labeled with their types—helix, sheet, and loop. Edges are labeled with lengths in amino acids.

(5) **Synthetic** is artificially generated data by GraphGen (http://www.cse.ust.hk/graphgen/) GraphGen creates a collection of labeled, undirected and connected graphs. It allows us to specify various parameters such as data size, the average graph density, graph size, and the number of distinct vertex labels; e.g., Syn10K.E30.D10.L5 dataset contains 10K graphs; the average size of each graph is 30; the density of each graph is 10%; and the number of distinct vertex and edge labels are 5 and 2, resp. A number of synthetic datasets are used in the experiments in order to see the performance changes with varying density values.

Query sets: 100 graphs were randomly selected from each dataset as its query graphs.

Due to the hardness of GED computations, small subsets of AIDS graphs were used for testing the GESS query methods in [7,9,10,16,24,34], whereas the entire Linux dataset was only used for testing the filtering power of methods in [35,36]. Chem_1M was recently used for testing graph edit similarity joins in the cloud [37]. To carry out a comparative study on our machine, we chose 10K AIDS graphs, 100K Chem_1M graphs, and put 10K s as time limit for each algorithm to run. Besides, we ran CSI_GED on the whole real data; results are shown in the scalability study.

7.1. Evaluation against graph order

In these experiments we compare the performance of CSI_GED with the recent vertex-based method DF-GED [25] and the state-of-art A^*-based method [15]. These experiments were performed against graph order to show how large the graph would be for each algorithm to work with on our machine. Six groups of three graphs each were randomly selected from each dataset such that the graphs in each group have consecutive graph order in the range: 8 ± 1, 11 ± 1, 14 ± 1, 17 ± 1, 20 ± 1, and 23 ± 1, resp.

Fig. 11 plots the average running time of each algorithm on each group, where the graphs in each group are compared with each other in a self-join manner. It also shows the enumeration time taken by CSI_GED algorithm against graph order, represented by CSI_GED-Enum. The figure shows that A^* and DF-GED fail to run on higher-order groups, i.e., on groups consisting of graphs with order well beyond 12 vertices for A^* and 15 vertices for DF-GED, for all datasets. The failure of A^* is attributed to the lack of memory — 4 GB of physical memory is not enough to store the huge number of partial vertex maps needed by A^*. Though the memory footprint of DF-GED is low, it does not show better performance and much scalability over A^*. DF-GED's failure is attributed to the huge time required by upper bound computations. In contrast, the low memory requirements and the efficient pruning capabilities allow CSI_GED to run efficiently in any computational environment. Moreover, on groups where A^* and DF-GED can run, CSI_GED significantly outperforms both competitors. It starts with 2–3 orders of magnitude performance gap and increases with graph order to become over three orders of magnitude. One of the main reasons of the increasing performance gap is that A^* and DF-GED explore exponentially growing space of vertex maps with respect to graph order. Finally, the CSI_GED-Enum curve shows that much of the CSI_GED's time is consumed by the map enumeration process. Thus, we can conclude that CSI_GED is a highly efficient algorithm for computing the edit distance on small graphs and scales gracefully with larger ones.

7.2. Effect of heuristics

7.2.1. Effect on processing time

In order to show the influence of the different heuristics on the performance of CSI_GED, we injected these heuristics one by one into the base algorithm and monitored the speedup achieved by each heuristic. We use the term "Basic" for the baseline algorithm without applying any heuristics. "+h_1" denotes the improved version of Basic by incorporating the first heuristic (Section 5.1). "+h_2" denotes the improved version of +h_1 by incorporating the second heuristic (Section 5.2). "+h_3" denotes the improved version of +h_2 by incorporating the third heuristic (Section 5.3). These experiments are carried out in the context of graph edit similarity search. Fig. 12 plots the time of each algorithm version at different τ on the different datasets. It is shown that Basic is unable to finish within the time limit on AIDS_10K at $\tau > 4$, on Chem_100K at $\tau > 3$ and on the synthetic datasets at $\tau > 6$. It is also shown that

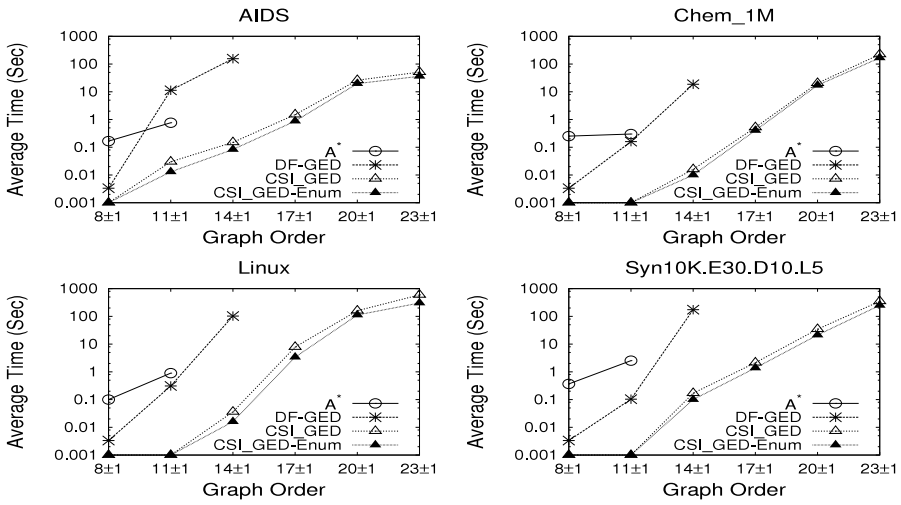

Fig. 11. Performance comparison with A^* algorithm against graph order.

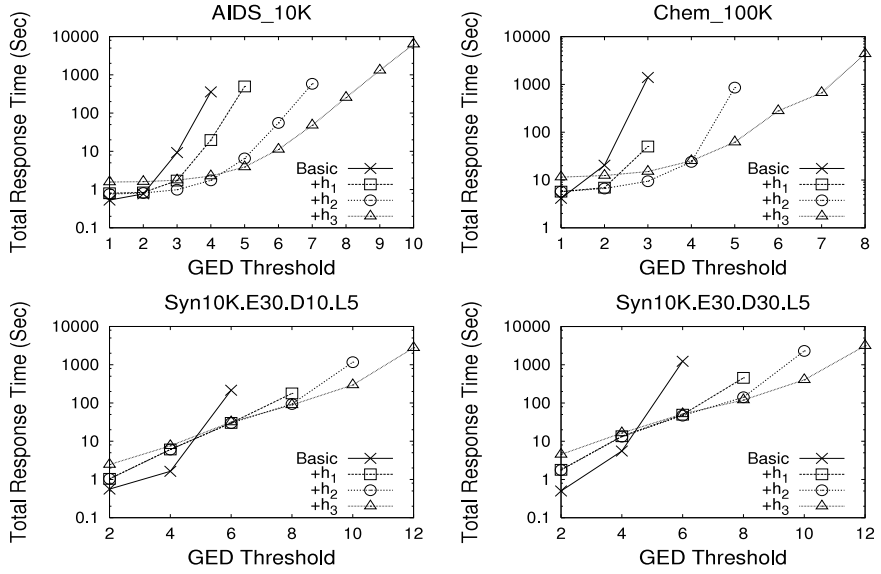

Fig. 12. Effect of heuristics on the performance of `CSI_GED`.

each heuristic enabled our algorithm to finish at larger τ within the time limit. For example, on AIDS_10K, +h_1 could finish at $\tau = 5$, +h_2 could finish at $\tau = 7$ and +h_3 could finish at $\tau = 10$. Moreover, the time of +h_3 at $\tau = 8$ is better than that of `Basic` at $\tau = 4$.

The speedup achieved using each heuristic is clear from the figure; +h_3 brought around 100x speedup over `basic` on Chem_100K at $\tau = 3$, 200x on AIDS_10K at $\tau = 4$, 7x on Syn10K.E30.D10.L5 at $\tau = 6$ and 26x on Syn10K.E30.D30.L5 at $\tau = 6$. Even though there is no speedup by +h_3 over +h_2 at small τ, the performance gap becomes clear at $\tau > 4$ on the real datasets and at $\tau \geq 6$ on the synthetic ones. This performance boost from `basic` to +h_3 is attributed to the efficient and effective pruning strategies.

7.2.2. Effect on search space

We also estimated effects of the different heuristics on the search space. In this experiment, we counted the number of investigated edge maps before and after using each heuristic. By this we can show how many maps have been excluded from computations by employing a given heuristic. As previously, these heuristics have been applied in succession, one after another, to see how much it cuts the search space in addition to the previous ones.

Fig. 13 plots the average number of edge maps investigated by each algorithm version on the different datasets at different τ. The figure shows that each heuristic has reduced the search space posed by the previous algorithm version. The space cut is remarkable for the second and third heuristics on the real data and for the first heuristic on synthetic data. Notice, for example on the AIDS dataset, that the search space of +h_3 at $\tau = 4$ is 10^4 times smaller than that of `basic` and 10^3 times smaller than that of +h_2. The same trend occurs on the Chem_100k dataset. For the synthetic dataset, +h_1 has the most impact.

7.3. Effect of label indexing on lookahead

The data plotted for ("+h_3") in Figs. 12 and 13 include the base lookahead pruning with the label indexing improvement (Section 5.3.1). Here, in this set of experiments, we quantify how much improvement is brought to the lookahead pruning by label indexing. For this purpose, we show the effects on the performance of +h_3 w.r.t. both edit thresholds and the number of edge labels. Two variant of +h_3 are used in the experiments labeled as "+h_3 with label indexing" and "+h_3 without label indexing".

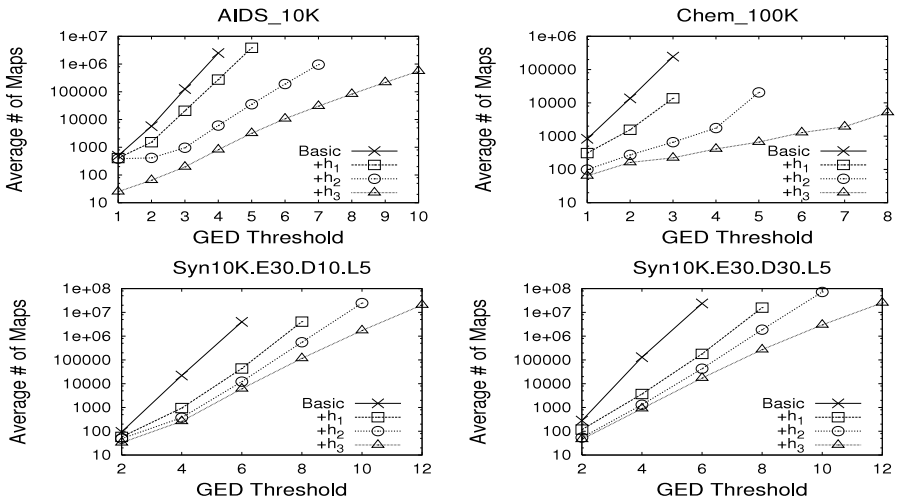

Fig. 13. Effect of heuristics on reducing the search space of CSI_GED.

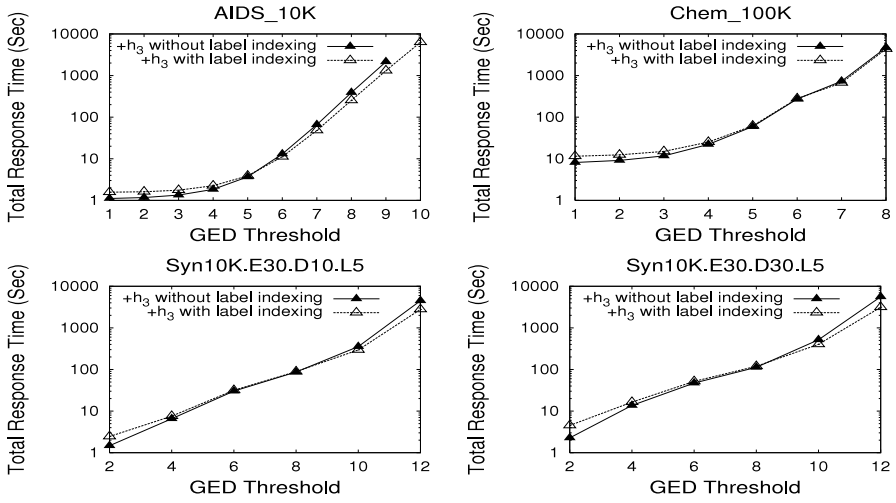

Fig. 14. Effect of label indexing: different thresholds.

7.3.1. *Effect w.r.t. edit threshold*

Fig. 14 plots the total response time of $+h_3$ variants on the different datasets and at different τ. The figure shows that at small values of τ, "$+h_3$ without label indexing" beats "$+h_3$ with label indexing" on all datasets. However, on larger τ, label indexing gradually increases the lookahead performance. On AIDS, for example, lookahead with label indexing enabled CSI_GED to finish within the time limit at $\tau = 10$, which could not be possible without label indexing. The initial degradation brought by label indexing is due to the preprocessing time taken on preparing the look-up tables. Fortunately, this time is in seconds and does not affect the overall performance, and this preprocessing time is well rewarded at higher τ.

7.3.2. *Effect w.r.t. # of labels*

Fig. 15 shows the total response time of $+h_3$ variants on the synthetic datasets when edge labels are increased in each synthetic dataset from 6–10 labels and the edit threshold is fixed at ($\tau = 16$). It is clear from the figure that lookahead with label indexing enabled CSI_GED to finish within the time limit when there is only 6 edge labels, which is not possible without label indexing. Also, at higher values of labels, there is a big improvement; it is about a factor of 3 performance improvement.

7.4. *Evaluating the first upper bound of* CSI_GED

7.4.1. *Approximation quality and computation time*

In this experiment, in order to test the quality of the first upper bound obtained by CSI_GED, abbreviated as FUB, we compare it against the one obtained by the state-of-the-art upper bound computation methods such as Assignment Edit Distance (AED) method [29]. The comparison was done to show how far is the distance value produced by each method from the exact edit distance GED.

To perform this experiment, 50 random graphs were selected from each dataset, on which self-join is applied using each method. For each method the mean relative overestimation of the exact graph edit distance (ϕ_o)[6] is calculated. Obviously, the smaller ϕ_o is, the better (i.e. nearer to the exact distance) is the approximation. We also aim at investigating the mean run time ϕ_t in order to correlate the distance accuracy achieved by each method with run time. We use a publicly available Java implementation of AED method [15].[7]

6 In [38], ϕ_o is defined for a pair of graphs G_1 and G_2 as: $\phi_o = \frac{|\lambda - GED|}{GED}$, where λ and *GED* are the approximate and exact graph edit distances between G_1 and G_2, resp.

7 http://www.fhnw.ch/wirtschaft/iwi/gmt.

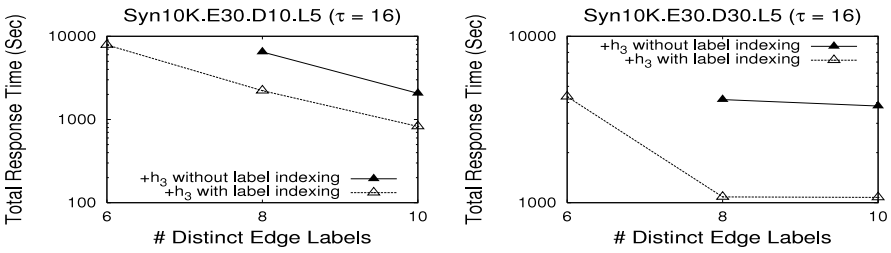

Fig. 15. Effect of label indexing: different number of edge labels.

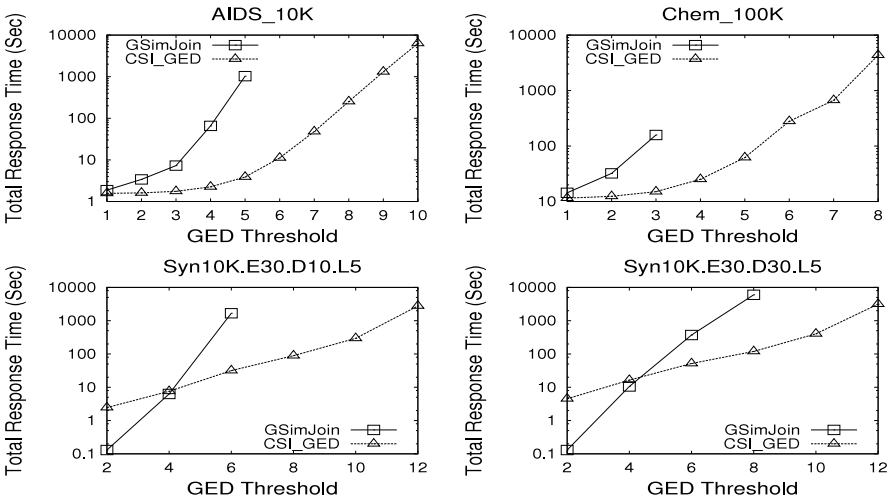

Fig. 16. Performance comparison with `GSimJoin` against edit threshold.

Table 1
ϕ_o and ϕ_t using the exact and approximation algorithms.

Dataset/edit cost	AIDS		Chem_1M		SynD10		SynD30	
	ϕo	ϕt	ϕo	ϕt	ϕo	ϕt	ϕo	ϕt
GED	0%	1.89	0%	0.006	0%	18.5	0%	0.56
FUB	32.52%	0.00024	43.86%	0.00014	38.19%	0.00036	44.41%	0.00032
AED	68.13%	0.00156	78.24%	0.00102	88.31%	0.00256	54.53	0.00180

Table 1 shows ϕ_t and ϕ_o of FUB, AED and GED values for the different datasets. First, it is clear that $\phi_o = 0$ for GED. For the others, FUB shows smaller losses in accuracy than AED. The accuracy losses of FUB are almost half of those produced by AED on most of the datasets. The only exception is SynD30 dataset, where the two methods have closed accuracy losses; 44% for FUB compared to 54% for AED. These results confirm that `CSI_GED` always produces FUB values which are closer to the exact values than those produced by AED on all datasets. In addition to the good results on FUB tightness, the average run time ϕ_t of `CSI_GED` to get FUB values is better than that with AED method; it is about 7 times faster. In conclusion, we can see that `CSI_GED` initially provides tight upper bound values at a very good response time compared with the current overestimation methods.

7.4.2. Classification accuracy

In this experiment we show the impact of the first overestimation (FUB) produced by `CSI_GED` on the task of graph classification. Two real datasets, AIDS and Protein, are used in this experiment. In AIDS two classes are distinguished, namely active and inactive, which represent the activity of the chemical compounds against HIV. In Protein, six classes (EC 1, EC 2, EC 3, EC 4, EC 5, EC 6) are identified, corresponding to enzyme class labels from the BRENDA enzyme database.[8] For the task of graph classification, we randomly selected 100 graphs from each dataset for training. For testing 250 AIDS graphs and 100 Protein graphs are used. The AIDS and Protein datasets with class labels are obtained from the IAM graph database.[9]

Graph classification is performed with respect to the k-nearest neighbor classifier (KNN) based on the approximate distances FUB and AED, resp. Nearest neighbors are selected in the training set and the independent test set is used to measure the classification accuracy. Finally, the number of nearest neighbors has been set in the experiments to $k = 1, 3, 5, 7$.

Table 2 shows the classification accuracy achieved at different values of k using the two GED estimates. First, it is worth noticing that changing the value of k does not affect the accuracy achieved on the AIDS dataset using both GED estimates. On the contrary, on the Protein dataset the classification accuracy deteriorates as k increases. Also, the classification accuracy is comparable for most k values using both GED estimates on the Protein dataset. On the other hand, on AIDS, the accuracy is better with our method than with AED method at all k values, approaching the largest difference in accuracy, $\triangle = 16\%$, at $k = 3$ for the benefit of our method.

[8] www.brenda-enzymes.org.
[9] http://www.fki.inf.unibe.ch/databases/iam-graph-database.

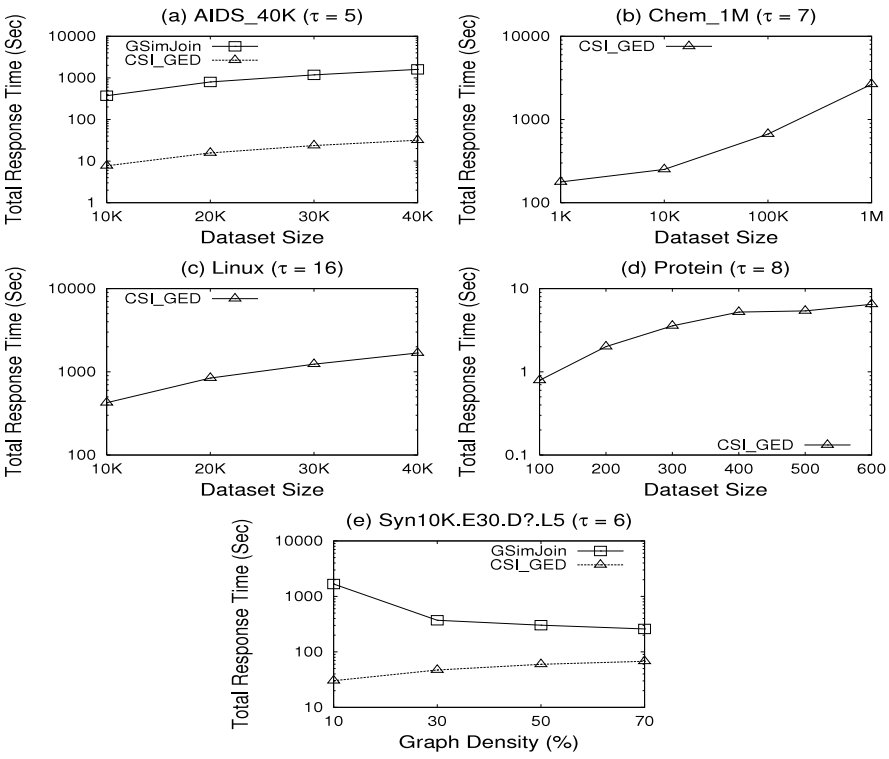

Fig. 17. Scalability on (1) dataset cardinality [a–d]; and (2) graph density [e].

Table 2
Classification accuracy in percentage, and accuracy difference △ of FUB when compared with AED.

Dataset/edit cost	AIDS				Protein			
	$k=1$	$k=3$	$k=5$	$k=7$	$k=1$	$k=3$	$k=5$	$k=7$
FUB	95.6%	97.6%	96%	94%	25.5%	13.5%	11.5%	9.5%
AED	84%	81.6%	81.2%	80.4%	25%	16.5%	11%	9%
△	+11.6%	+16%	+14.8%	+13.6%	+0.5%	−3%	+0.5%	+0.5%

7.5. Evaluation as a GESS query method

Here, we evaluate CSI_GED as a GESS query method. To do so, we compared CSI_GED with the state-of-the-art, indexing-based GESS methods such as GSimJoin [10,16] on real and synthetic graphs by varying τ.[10] GSimJoin is a path-based q-gram approach [10,16]. It filters data graphs based on the number of matching q-grams with those of the query, as well as a global lower bound on GED. GSimJoin uses indexing to accelerate bounds' computations. It also uses an improved version of A^* as a verifier. The executable of GSimJoin was obtained from their authors. As q-gram based approaches, the performance of GSimJoin is influenced by the gram size. For the real data, the best performance was achieved when $q = 4$, and when $q = 1$ for the synthetic ones. This variance is attributed to the fact that graph density influences the number of path-based q-grams. The greater the graph density, the more path-based q-gram in a graph.

Fig. 16 shows the effect of increasing τ on the performance of algorithms. It reports the total response time of each algorithm at different τ. If there is no plotted data for an algorithm at some τ values, it means the algorithm could not finish within the time limit on our machine for that value. CSI_GED shows the best performance on all datasets. For τ values where GSimJoin can finish, CSI_GED outperforms GSimJoin by over two orders of magnitude on the real datasets, and by up to two orders of magnitude on the synthetic ones. On synthetic data, GSimJoin starts faster at smaller τ, then both algorithms become comparable at $\tau = 4$. For larger τ, CSI_GED beats GSimJoin, and the performance gap increases with τ. In fact, the filtering quality and time of GSimJoin are not the main reasons for this low performance, A^* performance is another reason. Even though a far less graph matching operations are performed in the candidate verification phase of GSimJoin, the low performance of A^* prevents GSimJoin from being competitive with CSI_GED. On the other hand, CSI_GED implicitly uses effective lower and upper bounds as well as ordering and pruning strategies to quickly confine the search space.

7.6. Evaluating scalability

In order to test the scalability of CSI_GED against the dataset cardinality, we ran CSI_GED and GSimJoin on different subsets of the real datasets. Fig. 17(a–d) shows the total response time of both algorithms on the generated subsets of AIDS at $\tau = 5$, of Chem_1M at $\tau = 7$, of Linux at $\tau = 16$ and of Protein at $\tau = 8$, resp. Since GSimJoin failed to run on Chem_1M, Linux and Protein at the chosen thresholds, we could not report on its scalability for these datasets. On AIDS dataset (Fig. 17(a)), where GSimJoin can

[10] Other GESS methods do exist, such as SEGOS [35], Pars [24], Mixed [34] and ML-Index [39]. Experimentation in [10,16,24] revealed that GSimJoin outperforms SEGOS and is slightly outperformed by Pars. For instance, Pars is reported to be 3x faster than GSimJoin on a small subset of AIDS. Also, experimentation in [39] show ML-Index slightly outperforms Pars. Our experimentation show that CSI_GED is 330x faster than GSimJoin on AIDS_10K at $\tau = 5$. Although the executable of Mixed [34] is available, it is excluded from comparisons because it uses an approximated GED verifier and therefore its final results are not precise.

run, the two methods are not very sensitive to this parameter, and the running time grows slowly. However, CSI_GED shows the best performance. The performance gain is consistent with the previous experiments. Fig. 17(b)–(d) show that CSI_GED scales gracefully on the other real datasets.

Fig. 17(e) shows the effect of changing the density of synthetic graphs on the performance of algorithms at $\tau = 6$. It shows CSI_GED scales gracefully with this parameter. GSimJoin, on the other hand, shows less sensitivity, and the performance gap with CSI_GED decreases with increasing density. In fact, this efficiency is brought to GSimJoin by A^*. Since the size of synthetic graphs is fixed to 30 edges each, the graph order decreases with increasing density; it is about 6 vertices at higher density. A^* is efficient on comparing small and dense graphs.

8. Conclusions

For a long time graph edit distance computation are performed using the search paradigm A^*. Existing A^*-based methods have shown inability to compare large graphs. To enable comparison on larger and distance graphs, this paper introduced CSI_GED, a novel edge-centric approach for computing graph edit distance through common sub-structure isomorphism enumeration. CSI_GED utilizes backtracking combined with efficient heuristics to quickly search the edge-mapping space for those inducing optimal edit paths. Experiments showed that CSI_GED is highly efficient for computing graph edit distance; it outperforms the state-of-the-art methods by over three orders of magnitude. It is also shown that CSI_GED scales the computation gracefully to larger and distant graphs on which current methods fail to run. Moreover, CSI_GED is evaluated as a stand-alone graph edit similarity search query method. The experiments showed that CSI_GED is effective and scalable, and outperforms the state-of-the-art indexing-based methods by over two orders of magnitude.

References

[1] D. Conte, P. Foggia, C. Sansone, M. Vento, Thirty years of graph matching in pattern recognition, Int. J. Pattern Recognit. Artif. Intell. 18 (2004) 265–298.
[2] A. Khan, Y. Wu, C. Aggarwal, X. Yan, Nema: Fast graph search with label similarity, PVLDB 6 (2013) 181–192.
[3] N.H. Pham, H.A. Nguyen, T.T. Nguyen, J.M. Al-Kofahi, T.N. Nguyen, Complete and accurate clone detection in graph-based models, in: ICSE, 2009, pp. 276–286.
[4] K. Borgwardt, C. Ong, S. Schnauer, S. Vishwanathan, A. Smola, H.-P. Kriegel, Protein function prediction via graph kernels, Bioinformatics 21 (2005) 47–56.
[5] S. Melnik, H. Garcia-Molina, E. Rahm, Similarity flooding: A versatile graph matching algorithm and its application to schema matching, in: ICDE, 2002, pp. 117–128.
[6] J. Raymond, E. Gardiner, P. Willett, Rascal: Calculation of graph similarity using maximum common edge subgraphs, Comput. J. 45 (6) (2002) 631–644.
[7] Z. Zeng, A. Tung, J. Wang, J. Feng, L. Zhou, Comparing stars: On approximating graph edit distance, PVLDB 2 (1) (2009) 25–36.
[8] H. He, A. Singh, Closure-tree: An index structure for graph queries, in: ICDE, 2006, pp. 38–49.
[9] G. Wang, B. Wang, X. Yang, G. Yu, Efficiently indexing large sparse graphs for similarity search, IEEE Trans. Knowl. Data Eng. 24 (3) (2012) 440–451.
[10] X. Zhao, C. Xiao, X. Lin, W. Wang, Efficient graph similarity joins with edit distance constraints, in: ICDE, 2012, pp. 834–845.
[11] A. Robles-Kelly, E. Hancock, Graph edit distance from spectral seriation, IEEE Trans. Pattern Anal. Mach. Intell. 27 (3) (2005) 365–378.
[12] M. Neuhaus, H. Bunke, Edit distance-based kernel functions for structural pattern classification, Pattern Recognit. 39 (2006) 1852–1863.
[13] P. Hart, N. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Trans. Syst. Sci. Cybern. 4 (2) (1968) 100–107.
[14] K. Riesen, S. Fankhauser, H. Bunke, Speeding up graph edit distance computation with a bipartite heuristic, in: MLG, 2007, pp. 21–24.
[15] K. Riesen, S. Emmenegger, H. Bunke, A novel software toolkit for graph edit distance computation, in: GbRPR, 2013, pp. 142–151.
[16] X. Zhao, C. Xiao, X. Lin, W. Wang, Y. Ishikawa, Efficient processing of graph similarity queries with edit distance constraints, VLDB J. 22 (2013) 727–752.
[17] E. Schadt, S. Friend, D. Shaywitz, A network view of disease and compound screening, Nat. Rev. Drug Discov. 8 (4) (2009) 286–295.
[18] K. Gouda, M. Hassaan, CSI_GED: An efficient approach for graph edit similarity computation, in: ICDE, 2016, pp. 265–276.
[19] H. Bunke, On a relation between graph edit distance and maximum common subgraph, Pattern Recognit. Lett. 18 (8) (1997) 689–694.
[20] L. Brun, B.B. Gaüzère, S. Fourey, Relationships between graph edit distance and maximal common structural subgraph, Technical Report (GREYC), 2012, http://hal.archives-ouvertes.fr/hal-00714879.
[21] W. Zheng, L. Zou, X. Lian, J. Yu, S. Song, D. Zhao, How to build templates for rdf question/answering — an uncertain graph similarity join approach, in: SIGMOD, 2015, pp. 1809–1824.
[22] E. Krissinel, K. Henrick, Common subgraph isomorphism detection by backtracking, Softw.-Pract. Exp. 34 (2004) 591–607.
[23] F. Abu-Khzam, N. Samatova, M. Rizk, M. Langston, The maximum common subgraph problem: Faster solutions via vertex cover, in: AICCSA, 2007, pp. 367–373.
[24] X. Zhao, C. Xiao, X. Lin, Q. Liu, W. Zhang, A partition based approach to structure similarity search, PVLDB 7 (3) (2013) 25–36.
[25] Z. Abu-Aisheh, R. Raveaux, J.Y. Ramel, P. Martineau, An exact graph edit distance algorithm for solving pattern recognition problems, in: ICPRAM, 2015, pp. 271–278.
[26] L. Chang, X. Feng, X. Lin, L. Qin, W. Zhang, Efficient graph edit distance computation and verification via anchor-aware lower bound estimation, 2017, https://arxiv.org/abs/1709.06810.
[27] X. Chen, H. Huo, J. Huan, J.S. Vitter, Fast computation of graph edit distance, 2017, https://arxiv.org/abs/1709.10305.
[28] D. Blumenthal, J. Gamper, Exact computation of graph edit distance for uniform and non-uniform metric edit costs, in: GbRPR, 2017, pp. 211–221.
[29] K. Riesen, H. Bunke, Approximate graph edit distance computation by means of bipartite graph matching, Image Vis. Comput. 27 (7) (2009) 950–959.
[30] D. Justice, E. Hero, A binary linear programming formulation of the graph edit distance, IEEE Trans. Pattern Anal. Mach. Intell. 28 (8) (2006) 1200–1214.
[31] K. Gouda, M. Arafa, T. Calders, Bfst_ed: A novel upper bound computation framework for the graph edit distance, in: SISAP, 2016, pp. 3–19.
[32] K. Gouda, M. Arafa, T. Calders, A novel hierarchical-based framework for upper bound computation of graph edit distance, Pattern Recognit. 80 (2018) 210–224.
[33] J. Munkres, A network view of disease and compound screening, J. Soc. Ind. Appl. Math. 5 (1957) 32–38.
[34] W. Zheng, L. Zou, X. Lian, D. Wang, D. Zhao, Efficient graph similarity search over large graph databases, IEEE Trans. Knowl. Data Eng. 27 (4) (2015) 964–978.
[35] X. Wang, X. Ding, A.K.H. Tung, S. Ying, H. Jin, An efficient graph indexing method, in: ICDE, 2012, pp. 210–221.
[36] K. Gouda, M. Arafa, An improved global lower bound for graph edit similarity search, Pattern Recognit. Lett. 58 (2015) 8–14.
[37] Y. Chen, X. Zhao, C. Xiao, W. Zhang, J. Tang, Efficient and scalable graph similarity joins in mapreduce, Sci. World J. 2014.
[38] A. Fischer, C. Suen, V. Frinken, K. Riesen, H. Bunke, Approximation of graph edit distance based on Hausdorff matching, Pattern Recognit. 48 (2) (2015) 331–343.
[39] Y. Liang, P. Zhao, Similarity search in graph databases: A multi-layered indexing approach, in: ICDE, 2017, pp. 783–794.

Karam Gouda is a Professor at the department of Information Systems, Benha University, Egypt. He received his Ph.D. in computer science from Kyushu University, Japan, in 2002. His research interest spans the areas of data mining, graph matching and graph data management.

Mosab Hassaan received the Ph.D. degree in computer science from Benha University in 2013. He is a lecturer of computer science at the Faculty of Science, Benha University. His current research interests include data mining and databases.