CSI_GED: An Efficient Approach for Graph Edit Similarity Computation

Karam Gouda
Faculty of Computers & Informatics
Benha University, Egypt
Email: karam.gouda@fci.bu.edu.eg

Mosab Hassaan
Faculty of Science
Benha University, Egypt
Email: mosab.hassaan@fsc.bu.edu.eg

***Abstract*—Graph similarity is a basic and essential operation in many applications. In this paper, we are interested in computing graph similarity based on edit distance. Existing graph edit distance computation methods adopt the best first search paradigm A^*. These methods are time and space bound. In practice, they can compute the edit distance of graphs containing 12 vertices at most. To enable graph edit similarity computation on larger and distant graphs, we present `CSI_GED`, a novel edge-based mapping method 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 the edit distance on small as well as large and distant graphs. Furthermore, 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.**

I. Introduction

Large scale graph data are currently prevalent in domains such as Pattern Recognition, Bio-informatics, Chem-informatics, Social Networks, Semantic Web, Software Engineering, etc. Due to the widespread applications of graph models, a big research effort has been dedicated recently to various problems in managing and analyzing graph data.

Computing the similarity of graph objects is a basic and essential operation in many applications, including graph classification and clustering [1], [2], molecule comparison in chemistry [3], object recognition in computer vision [4], graph similarity search and join [5], [6], [7], [8], etc. In this paper, we are interested in computing graph similarity based on edit distance. Graph edit distance (GED) as a similarity measure is preferred over other distances or similarity measures because of its generality and broad applicability. It is applicable to virtually all types of data graphs with unconstrained label alphabets for both nodes and edges, and captures precisely structural differences. More interestingly, the accompanying edit sequence provides an explanation for an edit distance value and this is a very valuable feature for the user.

Unfortunately, the appealing properties of graph edit distance come at the cost of high computational complexity. Computing graph edit distance is known to be NP-hard problem [6]. Unlike other intractable graph matching problems such as subgraph isomorphism and maximum common subgraph isomorphism, to compute graph edit distance, a vertex of one graph has a possibility to be mapped to any vertex of the other graph regardless of their labels and degrees. Therefore, the search space is exponential with respect to the number of vertices of the involved graphs.

Very little work has been presented to address the high complexity of graph edit distance computation. Most of the existing methods adopt the best first search paradigm A^* [9], [10], [11], [12]. The basic idea of A^* is to find a map from the vertices of one graph to the vertices of the other graph, which induces a minimum edit cost. To achieve its goal, A^* explores the underlying vertex mapping space much like traversing an ordered tree, where intermediate tree nodes represent partial maps and leaf nodes represent complete ones. At each search state, A^* selects the best partial map to expand, where that map is the one with current minimum induced edit cost. This procedure continues until the selected map is a complete one.

The main problem with A^*-based methods is that the number of partial maps gets very large, especially when comparing large and distant graphs. Most of those maps cannot be discarded and have to be maintained until a very late stage of the search. As a consequence, huge memory is required. Another bottleneck is the expensive computation that is required for selecting the minimum-cost partial maps, and for updating that cost for each possible map extension. The memory and computational overhead make A^*-based methods unable to compute the edit distance of graphs having more than 12 vertices. 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 similarity edit threshold with the new compound. This step, called compound screening [13], helps the drug designer to get initial view of the compound at hand since similar compounds may have similar biological activities. The chemical compound database contains graphs with average order doubling at least the order of those that can be processed by A^*.

In this paper, we presents a novel approach for graph edit distance computation, called `CSI_GED`, which minimizes memory requirement and scales to larger and distant graphs. `CSI_GED` uses a completely different approach to get the edit distance. Instead of mapping vertices and then deducing the edit cost on edges, `CSI_GED` considers mapping edges first and the edit cost on their end vertices follows directly as a by-product. Even though the space of edge maps seems to be relatively large, edges are allowed to match only if their composing vertices are consistent with the previously matched

ones, called *common sub-structure isomorphism* restriction. This restriction reduces the search space sharply. Moreover, computing the induced edit cost of partial edge maps becomes easy and straightforward as it is directly calculated from the derived common sub-structures. In contrast, computing that cost with A^*-based methods is rather expensive, and is done in a separate phase for each possible map extension.

CSI_GED utilizes *backtracking* to explore the edge mapping space. The most important benefit is that, the memory requirement, a big burden for A^*-based methods, is diminished since CSI_GED enumerates edge maps in a depth first manner, which is efficient in memory consumption. Moreover, the framework of CSI_GED allows to implement a number of heuristics to quickly prune away a large portion of unpromising common sub-structure isomorphisms. These heuristics are developed based on the fact that the edit costs of previously explored complete edge maps, i.e. those seen so far in the search, are valid upper bounds on graph edit distance. Thus, the main goal of these heuristics is to enforce search states whose corresponding edit costs exceeding the minimum upper bound value to be encountered early; thus cutting large space from consideration. To achieve this objective, the first heuristic is to enable fast finding of tighter upper bounds, whereas the second maximizes the initial cost assigned to each map. The last heuristic implements look-ahead in the search.

Experiments show that CSI_GED is highly efficient for computing the edit distance on small as well as large and distant graphs. Furthermore, 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.

The remainder of this paper is organized as follows. The problem of graph edit similarity computation and state-of-the-art computation methods are presented in Section II. Section III presents the framework of CSI_GED and the motivation behind its construction. The different heuristics used to optimize CSI_GED are presented in Section IV. An application of CSI_GED to the graph edit similarity search problem appears in Section V. The experimental results are reported in Section VI. Related work is discussed in Section VII. Section VIII concludes the paper.

II. Preliminaries

A. Problem Definition

A graph G is defined as a pair of sets (V, E), where $V = \{v_1, v_2, \ldots, v_{|V|}\}$ is the set of vertices and $E \subseteq V \times V$ is the set of edges (directed or undirected). $|V|$ and $|E|$ are the numbers of vertices and edges in G, and are called the *order* and *size* of G, resp. Given a set of discrete-valued labels Σ, a labeled graph G is a triplet (V, E, l), where l is a labeling function l: $V \cup E \rightarrow \Sigma$. Let L_V and L_E denote the multi-sets of labels assigned to the vertices and edges of G, resp. This paper focuses on simple (no self-loops, no duplicate edges), undirected and labeled graphs. In what follows, an unlabeled version of a labeled graph G, i.e. its structure, is referred to as $S(G)$, and a labeled graph is simply called a graph unless stated otherwise.

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 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 $G \cong G'$.

A graph *edit operation* [14] is an operation on the graph to transform it to another one. Edit operations include insertion or deletion of vertices or edges, or the change of vertex or edge labels (called relabeling). 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 edit cost of transforming G_1 to G_2 using P is defined as: $C(G_1, G_2, P) = \sum_{i=1}^{k} c(p_i)$, where $c(p_i)$ is the cost of an individual edit operation p_i. Taking the unit cost for each edit operation, that is $c(p_i) = 1, \forall i$, an edit path of minimal length is called an *optimal edit path*.

Definition 2: **Graph edit distance (GED).** Given two graphs G_1 and G_2. The edit distance between G_1 and G_2, denoted as $GED(G_1, G_2)$, is the length of an optimal edit path transforming G_1 into G_2.

The following are two simple but effective lower bounds of GED, which we use throughout the paper. They are known as global (label) bounds. The first one is derived based on the differences in size and order of the comparing graphs, and given by [6] as:

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

The second bound improves the previous one by taking labels as well as structure information into account, and given by [8], [12] 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 any sets X and Y.

Definition 3: **(Maximum) common sub-structure.** 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. $|V'| > |V|$).

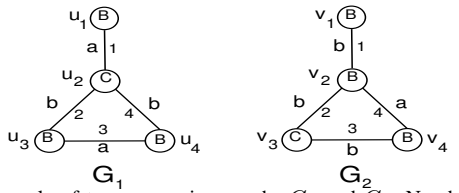

Fig. 1. Example of two comparing graphs G_1 and G_2. Numbers on edges are their ids, and each edge e_k is defined as: $e_k = (u_i, u_j)$ or $e_k = (v_i, v_j)$, $i < j$.

Example 1: Figure 1 shows two comparing graphs G_1 and G_2. $GED(G_1, G_2)$ = 2 since G_1 can be transformed to G_2

with a minimal number of two edit operations as follows: A deletion operation of the edge (u_1, u_2) and an insertion of a new edge (u_1, u_4) with label b. Based on Equation 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 substructure since it has 4 edges and 4 vertices.

Computing graph edit distance is NP-hard problem [6]. Very little work has been presented to address the high complexity of graph edit distance computation. We next overview the state-of-the-art GED computation methods and highlight their limitations. Hereafter, the comparing 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).

B. GED Computation: A^* Approach

The state-of-the-art methods for graph edit distance computation are based on the A^* paradigm, which explores all possible one-to-one vertex maps between the source and target graphs in a best-first fashion [9], [10], [11], [12]. A^* maintains a set of partial vertex maps with their induced edit cost. At each search state, it picks-up a partial map with the minimum induced edit cost to extend, where the unmapped target vertices as well as the *null vertex* – a dummy vertex with special label – are possible candidates for extension. To guide the selection process to the most promising partial maps, the edit cost associated with each partial map is then refined to include a heuristic estimate on the edit distance of the *remaining parts* – the unmapped edges and vertices of the two graphs. A^* guarantees that the first complete map picked up is the optimal one, provided that the heuristic estimate is a lower bound on the edit distance of the remaining parts. A vertex map is *complete* if the source and target vertices appear in that map, and *partial* otherwise. In cases where the search is ended, and there are some unmapped target vertices, to complete the map, vertex insertion is performed at the source graph for each unmapped target vertex.

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 the source vertices be processed in the order $(u_1, u_2, \ldots)$, $f(V_1) = \{f(u_1), \ldots, f(u_{i-1})\}$ be a partial map to extend, and $c(f)$ denote its associated edit cost. The cost $c(f)$ is defined as: $c(f) = g(f) + h(f)$, where $g(f)$ stands for the induced edit cost on the mapped vertices and their implied edge edit operations, and $h(f)$ is a lower bound on the edit cost of the remaining parts. The partial map f is extended one item at a time as the search space is traversed. For each possible value for the new extension $f(u_i)$, i.e. a value from $(V_2 \setminus f(V_1)) \cup \{v^n\}$, where v^n is a null vertex with $l_2(v^n) \notin \Sigma$, [1] a new partial map is constructed $f(V_1) = \{f(u_1), \ldots, f(u_{i-1}), f(u_i)\}$, and the new $c(f)$ is calculated (see below). If the map f is of size $|V_1|$ and there are some unmapped target vertices, to complete the map a vertex is inserted at the source graph for each unmapped target vertex, and $g(f)$ is modified to include the cost of the inserted vertices and their implied edge edit operations.

Algorithm `Update_PED` (Figure 2) updates $g(f)$ based on the recent extension $f(u_i)$. It first evaluates the edit cost

[1] Mapping u_i to v^n is equivalent to source vertex deletion.

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

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

Fig. 2. Updating the partial edit cost $g(f)$.

on the recently mapped vertex u_i (lines 1-2), and then on its implied edges (lines 3-10). The implied edge edit operations are calculated as: an edge incident on u_i and whose the other vertex is already matched, i.e. the edge $(u_j, u_i), j < i$, 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. [10] gives an estimation of the edit distance between the remaining parts via bipartite matching. [12] uses the global label lower bound on the remaining parts as a heuristic estimate.

A^*-based methods face a number of problems. First, the number of partial maps gets very large, especially when comparing large graphs. Most of those maps cannot be discarded and have to be maintained until a very late stage of the search. This happens because a current unpromising partial map, i.e., a map with high edit cost, has a chance to be extended in an advanced stage of the search. As a consequence, huge memory is required. Second, searching for a minimum-cost partial map to extend is expensive. This operation requires $O(log\ n)$ if priority queue is used, where n is the number of reserved partial maps. Thus, the major challenge is when the comparing graphs are large and distant. The large and distant the comparing graphs are, the larger the number of partial maps that need to be maintained and processed. Finally, updating the associated edit cost $c(f)$ of a partial map f is computationally expensive, and has to be done in a separate phase for each possible map extension.

Obviously, such problems crucially hamper the A^*-based methods to be used with real-world applications. To address these problems, in this paper, we propose a novel approach for graph edit distance computation, called `CSI_GED`, which minimizes memory requirement and scales to large and distant graphs. Next, we introduce the working principle of the new approach.

III. CSI_GED: A Novel GED computation approach

There are two main concerns that should be taken into account while developing an efficient GED computation algorithm. The first is to find a way to leverage the partial edit cost which is computed at every search state, and the second is to develop a traversing technique that continues searching without relying on full information of partial maps. In other words, the determination of an optimal edit path must avoid as many as possible the problems of A^* approach in order

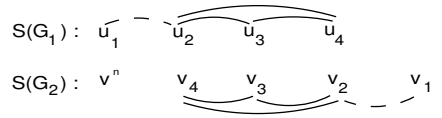

Fig. 3. The vertex map $f_3(V_1) = \{v^n, v_4, v_3, v_2\}$.

to scale to large and distant graphs. Below, we relate the GED computation problem to the problem of enumerating all common sub-structures in the comparing graphs.

Given two graphs $G_1 = (V_1, E_1, l_1)$ and $G_2 = (V_2, E_2, l_2)$, we next define the preserved edges under any vertex map f.

Definition 4: **Preserved edges of a vertex map.** Given a vertex map $f : V_1 \rightarrow V_2 \cup \{v^n\}$. An unlabeled source edge $(u, u') \in E_1$ is called preserved under f if $(f(u), f(u')) \in E_2$. An unlabeled target edge $(v, v') \in E_2$ is called preserved under f if $\exists\ (u, u') \in E_1$ 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. Consider the two sets of vertices associated with preserved edges, $V = \bigcup_{(u,u')\in E}\{u, u'\}$ and $V' = \bigcup_{(v,v')\in E'}\{v, v'\}$, also called preserved source and target vertices. Obviously, the two graphs $G = (V, E)$ and $G' = (V', E')$ have the same structure. Thus, the unlabeled 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 which is induced by f. The common sub-structure G can be disconnected and is not unique in the sense that different maps can determine it.

Example 2: Consider the comparing graphs G_1 and G_2 in Figure 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 $f_3(V_1) = \{v^n, v_4, v_3, v_2\}$. Figure 3 shows the map f_3. The preserved edges under this map are shown by the bold curves. Dashed curves show the unpreserved ones. The common sub-structure defined by f_3 is thus given by the preserved edges and their associated vertices as G''' = ($\{u_2, u_3, u_4\}$, $\{(u_2, u_3)$, (u_2, u_4), $(u_3, u_4)\}$). The common sub-structures defined by f_1 and f_2 are also given as $G' = S(G_1)$ and G'' = G'''. G' is both edge and node maximum since it has 4 edges and 4 vertices.

Theorem 1 restates the edit cost of any vertex map f in terms of its induced common sub-structure, and the unpreserved edges and vertices of the comparing graphs.

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 of G_1 and G_2 induced by f. Let $G^{l_1} \subseteq G_1$ 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 of f, $g(f)$, is given in terms of G as:

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

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: The map f induces an edit path P which transforms G_1 into G_2. The operations in P can be grouped into three sets of edit operations: edge deletion group D, vertrex/edge insertion group I and vertex/edge relabelling group R. The deletion group D consists of deleting all unpreserved source edges. These edges have no counterparts in the target graph. There are $|E_1| - |E|$ of such edges. The insertion group I consists of inserting vertices at the source graph corresponding to the unmatched target vertices, i.e. inserting $|V_2 \setminus f(V_1)|$ vertices. It also contains inserting edges into the source graph corresponding to the unpreserved target edges, i.e. inserting $|E_2| - |E|$ edges. The relabelling group of operations consists of relabelling on G^{l_1} and on the unpreserved source vertices $V_1 \setminus V$. The former is calculated as the number of corresponding vertices and edges of G^{l_1} and G^{l_2} which have different labels. This number is given as: $c_f(G^{l_1}, G^{l_2}) = |\{(u, u') \in E, l_1(u, u') \neq l_2(f(u), f(u'))\}| + |\{u \in V, l_1(u) \neq l_2(f(u))\}|$. The latter is calculated as the number of unpreserved source vertices having different labels from their corresponding target labels, i.e. $\Gamma(L_{(V_1 \setminus V)}, L_{(f(V_1) \setminus V)})$. ■

Algorithm 2: CSI_GED(G_1, G_2)

```
Enumerate all CSIs of G1 and G2;
for each CSI f do
    compute g(f) as in Equation 3;
    keep track of minimum g(f);
output the minimum g(f);
```

Fig. 4. CSI_GED approach for $GED(G_1, G_2)$.

Based on Theorem 1, it becomes easy and straightforward to compute the induced edit cost $g(f)$ of a complete vertex map f once its corresponding common sub-structure is identified. Example 3 shows that a common sub-structure of G_1 and G_2 can induce an edit cost less than that of a maximum one.

Example 3: Consider the comparing graphs G_1 and G_2 in Figure 1, and the three maps f_1, f_2 and f_3, defined in Example 2. The edit cost of transforming G_1 into G_2 using f_3 is equal to 8; it can be calculated in terms of the induced common sub-structure as: a deletion of the unpreserved source edge (u_1, u_2), a vertex insertion to correspond to the unmatched target vertex v_1, an edge insertion to correspond to the unpreserved target edge (v_1, v_2), 4 relabeling operations on the common sub-structure (two on the source vertices u_2 and u_3, and two on the source edges (u_2, u_4) and (u_3, u_4)), and a relabelling operation on the unpreserved source vertex u_1 (equivalent to u_1 deletion). Likewise, $g(f_1)$ = 5 and $g(f_2)$ = 2. Thus, the edit path induced by f_2 is the optimal one.

Note in Example 3 that although f_2 and f_3 generate the same common sub-structure, they induce different edit costs on that structure.

As such, a novel approach to compute graph edit distance 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 enumerated one the corresponding edit cost as in Equation 3. The graph edit distance is then calculated as the minimum of these costs. The approach, named CSI_GED (which stands for the bold letters in: Common Substructure Isomorphisms based Graph Edit Distance), is outlined in Figure 4, and its completeness is verified by Theorem 2.

Theorem 2: (Completeness) Given two comparing graphs G_1 and G_2. CSI_GED(G_1, G_2) returns the edit distance between G_1 and G_2.

The CSI_GED approach makes it possible to cut the

computation overhead of getting the edit cost, $g(f)$, of each vertex map f. Unfortunately, an equivalent computation cost is needed by any vertex-based mapping method enumerating CSIs. In such methods, see [15] for example, to construct a common sub-structure, a target vertex matches a source vertex if they do not violate the connections on the previously matched vertices. To check previous connections, the method exerts computations equivalent to that of calculating the implied edge edit operations as in A^*-based methods. Facing this challenge, `CSI_GED` constructs CSIs through mapping edges instead of vertices, that is, edge mapping instead of vertex mapping space is considered. Mapping edges eases, as we see next, the connection checking problem appearing while constructing CSIs. Although the edge mapping space seems, at first glance, to be relatively large, we next show that the space used for CSIs enumeration is considerably smaller than the full space. It could be smaller than the vertex-based mapping space on sparse graphs which is the case for many real-world applications.

A. CSIs Enumeration

Initially, for the sake of matching edges, we consider any target edge as an ordered pair of vertices. Thus, for any target edge $e = (v, v') \in E_2$, let $e^r = (v', v)$ denote its reverse and $\tilde{E_2} = \{e, e^r : e \in E_2\}$ denote 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 $e \rightarrow e'$, if and only if v and v' are matched with u and u', resp. Next, we give the properties that any edge map must satisfy when it comes to identify a common sub-structure.

Lemma 1: Given two comparing 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 the null edge, 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 one or more source edges to be mapped to the null edge; 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 connection vertex u.

Lemma 1 shows that the space of edge maps considered for CSIs enumeration is much smaller than the original space. In what follows, the edge map f that defines a common sub-structure 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` uses *backtracking* to traverse the edge mapping space for CSIs enumeration. Backtracking views edge maps as arranged in a tree-like structure, and works as follows. Initially, an edge map f is empty; it is extended one edge at a time, as the search space is traversed. The length of f is the same as the depth of the corresponding node in the search tree. Given a partial edge map of length l, $f_l = \{e_{i_0}, e_{i_1}, \ldots, e_{i_{l-1}}\}$, the possible values for the next extension e_{i_l} comes from a subset $C_l \subseteq P_l$ 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*. Figure 5 outlines the backtracking algorithm. The main loop tries extending the partial edge map f_l with every edge e' in the current combine set C_l. The first step (line 2) is to compute f_{l+1}, which is simply f_l extended with e'. e' and its reverse e'^r are then marked as matched at the second step (line 3). The third step (line 5) is to extract the new possible set of extensions, P_{l+1}, which consists only of target edges $e \in \tilde{E_2}$ that are not matched yet. The forth step (line 6) is to create a new combine set for the next pass, consisting of valid extensions. A target edge is a *valid* extension if it satisfies the second condition of Lemma 1, i.e. if its end vertices are consistent with the previously matched ones. The combine set, C_{l+1}, thus, consists of those edges in the possible set 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. The final step (line 7) is to recursively call `CSI-backtrack` for each extension. When matching all source edges (line 8), i.e. a complete map f is found, it is added to $\mathcal{CSI}$ – the set of all completed CSIs.

Algorithm 3: CSIs enumeration

```
1: CSI = ∅;                          /*a set to hold all CSIs*/
2: CSI-backtrack(∅, Ẽ2 ∪ {e^n}, 0);
3: return CSI;
CSI-backtrack(f_l, C_l, l)
1:  for each e' ∈ C_l
2:    f_{l+1} = f_l ∪ {e'};
3:    if e' ≠ e^n then mark e' and e'^r as matched;
4:    if l < |E1| − 1 then
5:      P_{l+1} = {e : e ∈ Ẽ2 and e is unmatched};
6:      C_{l+1} = CSI-combine(f_{l+1}, P_{l+1});
7:      CSI-backtrack(f_{l+1}, C_{l+1}, l + 1);
8:    else CSI = CSI ∪ {f};     /*f is a complete CSI*/
9:    f_{l+1} = f_{l+1} \ {e'};          /*restore state(lines 9-10)*/
10:   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})
1:  C = ∅;
2:  for each e ∈ P_{l+1}
3:    if e is a valid extension then C = C ∪ {e};
4:  return C ∪ {e^n};
```

Fig. 5. An edge backtracking algorithm for CSIs enumeration.

As presented, the backtracking method performs a depth-first traversal of the search space which offers efficient manipulation of memory, a major problem for A^*-based methods. Also, the validity of possible extensions can be easily checked by maintaining with each edge map f a vertex map M to store the mapping on the end vertices of already matched edges. When a new target edge (v, v') is considered for extending f on the source edge (u_i, u_j), a nonempty slot $M(i)$ or $M(j)$ must be equal to v or v', resp., to be a valid extension.

Theorem 3 gives an upper bound estimation on the size of the edge mapping space considered by `CSI_GED`.

Theorem 3: The search space considered by `CSI_GED` is of size $O(|E_2| \times (\frac{|V_2|}{2} - 2)! \times (d-1)^{|E_1| - \frac{|V_1|}{2}})$, where d is the maximum vertex degree of the target graph.

PROOF: First, without loss of generality, suppose G_1 and G_2 have even number of vertices each, and $|V_1| \geq |V_2|$. Since $\tilde{E_2} \cup \{e^n\}$ is the set of matching candidates for each edge $e_i \in E_1$, the total size of edge mapping search space is $\prod_{i=1}^{|E_1|} |\tilde{E_2}|$. However, the actual size of the search space considered by `CSI_GED` is calculated as $\mathcal{S} = \prod_{i=1}^{|E_1|} |C(e_i)|$, where $C(e_i) \subseteq \tilde{E_2}$ is the set of valid candidates in the search.

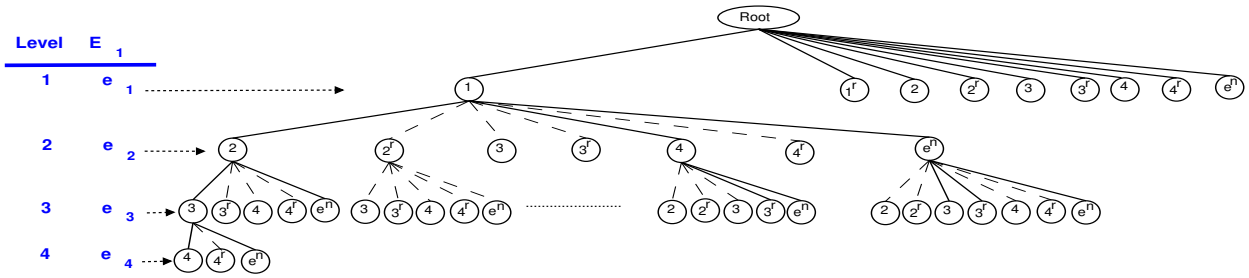

Fig. 6. Edge mapping/backtrack search tree. Black edges are considered by backtracking and dashed ones show pruned branches. e^n is a null edge.

The first source edge to be matched has a number of candidates equal to $|\tilde{E}_2| = 2 \times |E_2|$. In order to calculate the number of valid candidates for each remaining source edge $e_i \in E_1$, we first classify those edges, while searching, into two types: free and tied edges. An edge is called *tied*, denoted e^t, if it is adjacent to at least one previously matched edge, and called *free* and denoted e^f otherwise. Let F and T denote the sets of free and tied edges. Thus, $\mathcal{S} = \prod_{j=1}^{|F|} |C(e_j^f)| \prod_{k=1}^{|T|} |C(e_k^t)|$. The number of free edges is at most $\frac{|V_1|}{2}$, i.e. $|F| \leq \frac{|V_1|}{2}$, because any set of free edges, F, represents a maximal edge matching in G_1 with largest value is $\frac{|V_1|}{2}$ – the size of maximum edge matching. The number of valid candidates for each free edge is calculated as $\frac{|V_2|}{2} - k$, where k is the number of previously matched target edges and $\frac{|V_2|}{2}$ is the size of the maximum edge matching in G_2. Thus, $\prod_{j=1}^{|F|} |C(e_j^f)| = 2|E_2| \times \prod_{j=1}^{\frac{|V_1|}{2}-1} (\frac{|V_2|}{2} - 1 - j) = O(|E_2| \times (\frac{|V_2|}{2} - 2)!)$, because $|V_1| \geq |V_2|$ and $2|E_2|$ is the number of candidates of the first matched source edge. Considering tied edges, there are two cases. In the first case, the tied edge is adjacent to only one previously matched edge. In this case, there are at most $d - 1$ valid candidates, where $d = max_{v_j \in V_2}(deg(v_j))$. In the second case, the tied edge is connected to two previously matched edges. For each of such edges, there is only one valid candidate. Thus, $\prod_{k=1}^{|T|} |C(e_k^t)| \leq \prod_{k=1}^{|T|} (d-1) = (d-1)^{|T|}$. Since $|T|$ is at least $|E_1| - \frac{|V_1|}{2}$, then the space size is $\mathcal{S} = O(|E_2| \times (\frac{|V_2|}{2} - 2)! \times (d-1)^{|E_1| - \frac{|V_1|}{2}})$. ■

It is clear from Theorem 3 that the space considered by `CSI_GED` is much smaller than the vertex-based mapping space which is of size $O(|V_2|^{|V_1|})$, especially when the comparing graphs are sparse. It is because, in sparse graphs [2], the number of edges $|E_1|$ is very close to the number of vertices $|V_1|$, and $d \ll |V_2|$. Also, $(\frac{|V_2|}{2} - 2)! \ll (\frac{|V_2|}{2} - 2)^{\frac{|V_1|}{2}}$. Only when the target graph is a complete one which is of full density and $d = |V_2| - 1$, 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 $k = |V_2|$, the vertex mapping space is smaller.

Example 4: Consider the comparing graphs G_1 and G_2 in Figure 1. Figure 6 shows part of the full edge mapping search tree of G_1 and G_2, where nodes at level i indicate the target edges possible for matching the source edge $e_i \in E_1$. As the figure shows, a source edge is mapped to a target edge at a time according to the given order of the source edges. So, inner nodes correspond to partial edge maps and leave nodes correspond to complete ones. The backtrack search space can be considerably smaller than the full space. For example, we start with $f_0 = \emptyset$ 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 for e_1, P_1, consists of 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.

When comparing large graphs, the number of CSIs becomes quite large. Large search space is a major challenge for backtracking. To meet this challenge and develop an efficient `CSI_GED` algorithm, new heuristics are required to remove entire branches from the backtracking tree. We describe below the different heuristics used to optimize `CSI_GED`.

IV. Optimizing CSI_GED

Existing vertex-based backtracking algorithms for the maximum CSI problem use sub-structure size to prune the search space [15], [3], [16]. In our setting, however, sub-structure size has no role to play since we have to enumerate all common sub-structures. Fortunately, the map edit cost can be used instead for pruning the backtracking tree as follows.

Since backtracking enumerates CSIs in a depth first manner, some CSIs will be available early in the search before others. The edit cost induced by those enumerated ones are in fact upper bounds on the graph edit distance, and can be used to cut branches of the backtracking tree at some search states; precisely, at those tree nodes with (expected) higher edit costs. To do so, instead of calculating the edit cost induced by each CSI in a separate, subsequent phase as in Figure 4, we push the calculation (or a relevant part of it) into the CSIs enumeration process, and maintain a cost value for each partial CSI. If that value is no less than the current upper bound value, the search for those CSIs which extend the current partial CSI stops, and continues trying to extend other partial ones.

Before deciding on the relevant calculations that could be easily injected into the CSIs construction process, we need to show first what is contributing to the induced edit cost $g(f)$ of a given CSI f. The cost value $g(f)$ given by Theorem 1 is a summation of five independent values. The first one, $c_f(G^{l_1}, G^{l_2})$, calculates the edit cost 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 values calculate the number of unpreserved source and target edges, which are $|E_1| - |E|$ and $|E_2| - |E|$, resp. The last two values are obtained from the edit operations on

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

the source and target vertices that are not appearing in G. $|V_2 \setminus f(V_1)|$ vertex insertions for the unmatched target vertices are required, and this number constitutes the fourth value. The last value comes from the relabeling required on the unpreserved source vertices.

It is easy to inject the first two costs into CSIs construction. Given a search state, the common sub-structure identified at that state is expanded if a valid map extension is found; otherwise, it remains unchanged and the source edge is mapped to the null edge e^n, which means deleting that edge. Thus, on one hand, having the edit cost on matching the source and target edges used for extending f will increment $g(f)$ based on the first cost value. On the other hand, deleting the source edge will increment $g(f)$ based on the second cost value. The third and fifth costs could also be injected but at extra computations[3]; thus they remain to be calculated subsequently, i.e. after completing the map. An important part of the fourth value which is $||V_1|-|V_2||$ could be used as an initial cost for each CSI because it is global and independent of any CSI.

The new code for `CSI_GED` after cost injection is given in Figure 7. It is a straightforward extension of CSI-backtrack. The main addition is the injection of $g(f)$ on the first two values to eliminate branches of the backtracking tree. In addition to the main steps in CSIs enumeration, the new code starts with an hypothetical upper bound value $A = \infty$ and an initial cost value $IC = ||V_1|-|V_2||$ assigned to every CSI f. It adds a step (line 3, CSI-backtrack) after the map extension to update $g(f)$ and a step (lines 10-11, CSI-backtrack) to update A. To include pruning based on the upper bound value, a new condition is added to the main pruning step at line 3 of CSI-combine. The *edge matching cost* $emc(e \rightarrow e')$ that is used for updating $g(f)$ 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 $emc(e \rightarrow e')$, is given as:

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

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.

To boost pruning based on upper bounds, the idea that first comes to mind is that instead of pruning based on $g(f)$ alone, we can prune based on a cumulative of $g(f)$ and a lower bound on the edit distance of unmapped edges and vertices. However, it is not practical to compute a lower bound at every tree node of an ever expanding search tree. Here, new efficient heuristics are adopted to enhance pruning. The first heuristic arranges target edges in order to enable fast finding of tighter upper bounds, whereas the second maximizes the initial edit cost assigned to each CSI. The last heuristic implements look-ahead in the search. Such heuristics would allow tree nodes whose corresponding edit costs exceeding the upper bound values to be encountered early in the search; thus cutting many branches from consideration. Next, we detail each of these heuristics.

Algorithm 4: `CSI_GED`(G_1, G_2)

```
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(f_l, C_l, l, g(f_l))
1:  for each e' ∈ C_l
2:     f_{l+1} = f_l ∪ {e'};
3: *   g(f_{l+1}) = g(f_l) + emc(e_{l+1} → e');
4:     if e' ≠ e^n then mark e' and e'^r as matched;
5:     if l < |E1| − 1 then
6:        P_{l+1} = {e : e ∈ Ẽ2 and e is unmatched};
7:        C_{l+1} = CSI-combine(f_{l+1}, P_{l+1}, g(f_{l+1}));
8:        CSI-backtrack(f_{l+1}, C_{l+1}, l + 1, g(f_{l+1}));
9:     else  /*a complete CSI*/
10:*      if g(f_{l+1}) + |{e ∈ E2 : e is unmatched}| + Δ < A
11:*         A = g(f_{l+1}) + |{e ∈ E2 : e is unmatched}| + Δ ;
12:    f_{l+1} = f_{l+1} \ {e'};  /*restore state(lines 12-14)*/
13:    if e' ≠ e^n then mark e' and e'^r as unmatched;
14:*   g(f_{l+1}) −= emc(e_{l+1} → e');
CSI-combine(f_{l+1}, P_{l+1}, g(f_{l+1}))
1:  C = ∅;
2:  for each e ∈ P_{l+1}
3: *   if e is valid & g(f_{l+1}) + emc(e_{l+2} → e) < A
4:        C = C ∪ {e};
5:  return C ∪ {e^n};
```

Fig. 7. `CSI_GED` algorithm (* indicates a new line not in CSI-backtrack (Figure 5) and $\Delta = \lambda + |V_2 \setminus f(V_1)| - ||V_1| - |V_2||$).

A. Ordering Target Edges

Given that every valid extension of a size l, partial CSI comes from the same set of target edges $\tilde{E}_2$ (line 6, Figure 7). Thus, the target edges $\tilde{E}_2$ used at a tree level l, $1 \leq l \leq |E_1|$, could be ordered in such a way that those CSIs which produce tighter upper bounds would be enumerated first. The adopted ordering heuristic arranges $\tilde{E}_2$ at a tree level l in increasing order of a cost value $\mathcal{C}(e_l, e')$, where the function $\mathcal{C}$ computes an approximated graph edit cost, provided that the target edge e' is assigned to the source edge e_l. In order to define the cost function $\mathcal{C}$, we start by defining *edge star* – a local structure surrounding an edge – and *star matching cost.*

Definition 6: **Edge star.** Given a graph $G = (V, E, l)$ and an edge $e \in E$. 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 edges $e = (u, u')$ and $e' = (v, v')$, the source and target edges. 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 multiset of labels of edges incident on the vertex x, excluding the core edge label. [5]

[3] An easy injection of the third cost will be presented later in this section.

[4] Notice that since we are matching edges, 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.

[5] Note that the outer vertices surrounding each edge star are not involved in calculating the star matching cost.

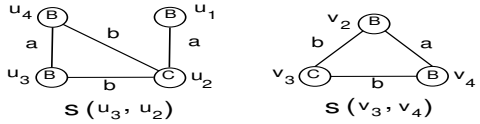

Fig. 8. Example of two edge stars $s(u_3, u_2)$ and $s(v_3, v_4)$.

Definition 8: **Star remaining graph.** Given a graph $G = (V, E, l)$ and an edge star $s(e)$, $e \in E$. we define the star remaining graph, denoted G^e, to be the graph obtained after removing the edge star $s(e)$ from G.

Given a 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 remaining graphs of the stars $s(e_l)$ and $s(e')$, resp. Now, we are ready to define the cost value $\mathcal{C}(e_l, e')$ to be equal to the star matching cost of $s(e_l)$ and $s(e')$ in addition to the vertex and edge label lower bounds of 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 comparing graphs G_1 and G_2 in Figure 1, the source and target edges $e = (u_3, u_2)$ and $e' = (v_3, v_4)$. Figure 8 shows the edge stars $s(e)$ and $s(e')$. The graphs $G^e = (\{u_1, u_4\}, \emptyset, l_1)$ and $G^{e'} = (\{v_1, v_2\}, \{(v_1, v_2)\}, l_2)$ are the stars' remaining graphs. The cost value $\mathcal{C}(e, e') = 5$, where the star matching cost is calculated as: $smc(e, e') = emc(e \to e') + \Gamma(LE_{u_3}, LE_{v_3}) + \Gamma(LE_{u_2}, LE_{v_4}) = 2 + 1 + 1 = 4$. The vertex and edge label bounds of the remaining graphs are calculated as 0 and 1, resp. If the source edges are processed 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 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 the cost values $\mathcal{C}$ are calculated w.r.t. the source edge star $s(u_2, u_3)$, and given as: $\{2, 2, 3, 4, 5, 5, 6, 6\}$.

B. Maximizing Initial CSI Cost

It is possible to maximize the initial cost value assigned to each CSI to include the difference in graph size as well as graph order difference. That is, the initial cost assigned to each CSI f could be refined to become $IC = ||V_1| - |V_2|| + ||E_1| - |E_2||$. Adding the graph size difference 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 the source edge deletion as of edit cost one (Def. 5), we modify it taking into account the different cases that may arise regarding the values $|E_1|$ and $|E_2|$. The following theorem covers these cases.

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 calculated as:

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

where k is the number of previously deleted source edges.

PROOF: We have two possible cases regarding the number of source and target edges, i.e. $|E_1|$ and $|E_2|$.

Case I: When $|E_1| \leq |E_2|$. To transform G_1 into a graph isomorphic to G_2, we initially need to add $|E_2| - |E_1|$ edges somewhere at the source graph G_1 to equalize edges of both graphs. Thus, during CSI construction, for each deleted source edge, a new edge is required to be added somewhere in the source graph, thus two edit operations, to keep an equal number of source and target edges.

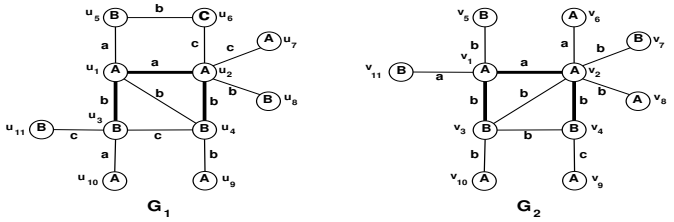

Fig. 9. Example of two comparing graphs G_1 and G_2.

Case II: When $|E_1| > |E_2|$. Initializing $g(f)$ with $|E_1| - |E_2|$ means that deleting up to this number of source edges has no effect on the edit cost since an equivalent cost is already added in the initialization. After exhausting the initial cost, i.e. after deleting $(|E_1| - |E_2|)$ source edges, any following source edge deletion requires another edge addition at the source graph, thus two edit operations, to keep an equal number of source and target edges. ■

In addition to maximizing the initial cost assigned to each CSI, Theorem 4 allows smooth injection of the edit cost of unpreserved target edges, $|E_2| - |E|$, into CSIs enumeration process. The pseudo-code of `CSI_GED` in Figure 7 is modified accordingly in order to accommodate the new definitions of edge matching cost and initial cost value. The number of unpreserved target edges is also removed from the subsequent calculations phase at lines 10 and 11 of `CSI-backtrack`.

C. Look-ahead Based Pruning

Consider the source and target graphs G_1 and G_2 given in Figure 9. These two graphs have the same number of vertices, but G_1 has an extra edge. Thus, the initial cost assigned to each CSI is equal to one. Consider a partial CSI f that matches the bold target edges with the bold source edges. The edit cost of this partial CSI, $g(f)$, remains equal to one since the edit cost on the subgraphs induced by the common sub-structure is equal to zero. For any upper bound value greater than one, it is not possible to stop extending this partial CSI at this stage based on the cost value $g(f)$. In this subsection 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, and see if it is larger than the current upper bound value to prune the map.

Given a graph G. We define for any 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 $G = (V, E, l)$ and a subgraph $H = (V_H, E_H, l) \subseteq G$. The inner neighborhood of V_H, denoted $N_I(V_H)$ is defined as: $N_I(V_H) = \{(u, v) \in E : u, v \in V_H\}$. The outer neighborhood of V_H, denoted $N_O(V_H)$ is defined as: $N_O(V_H) = \{(u, v) \in E : u \in V_H \wedge v \notin V_H\}$.

Based on the inner and outer neighborhoods of a subgraph $H \subseteq G$, we define *inner* and *outer degrees* for the vertices of H as follows. Let $N_I(V_H)$ and $N_O(V_H)$ be the inner and outer neighborhoods of V_H, the inner and outer degrees of a vertex $v \in V_H$, denoted $d_I(v)$ and $d_O(v)$ resp., are given as:

$d_I(v) = |\{(v,w) \in N_I(V_H)\}|$ and $d_O(v) = deg(v) - d_I(v)$, where $deg(v)$ is the vertex degree.

Now, given the source and target graphs G_1 and G_2, and a (partial) CSI f. Let M be the vertex map associated with the edge map f. The new cost function $g'(f)$ is defined in terms of the inner and outer neighborhoods of subgraphs as follows. Let $G^{l_1} \subseteq G_1$ and $G^{l_2} \subseteq G_2$ be the two subgraphs identified based on the f's common sub-structure $G = (V, E)$ of G_1 and G_2. The cost $g'(f)$ is calculated 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:

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

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:

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

where $(u,u') \in N_I(V_{G^{l_1}})$ and $(M(u), M(u')) \in N_I(V_{G^{l_2}})$, 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:

$$c_r = |n_1 - n_2|, \quad (6)$$

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

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

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

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

Example 6: Consider the graphs G_1 and G_2 in Figure 9. Let f be the current CSI that matches the bold target edges with the bold source edges, $M = \{v_1, v_2, v_3, v_4\}$ be the vertex map associated with f and G be the common sub-structure identified by f. The inner neighborhoods of G^{l_1} and G^{l_2} are given as: $N_I(V_{G^{l_1}}) = \{(u_1,u_2), (u_1,u_3), (u_1,u_4), (u_2,u_4), (u_3,u_4)\}$ and $N_I(V_{G^{l_2}}) = \{(v_1,v_2), (v_1,v_3), (v_2,v_3), (v_2,v_4), (v_3,v_4)\}$, resp. The outer neighborhoods are $N_O(V_{G^{l_i}}) = E_i \setminus N_I(V_{G^{l_i}})$, $i = 1, 2$. The inner and outer degrees of a vertex u_1, e.g., are given as: $d_I = 3$ and $d_O = 1$. 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 label bound cost as: $\kappa = max(11-9, 0+0) = 2$. Thus, if the current upper bound value is, e.g., 7, the subtree rooted at f could be pruned based on $g'(f)$.

Besides being a very effective tool for pruning the search space, evaluating the lookahead value $g'(f)$ for any partial CSI f is not computationally-demanding as the cost values c_d, c_e and c_r (Equations 4-6) are easy to compute, and the costly $\Gamma(L_{V_1}, L_{V_2})$ (Equation 7) could be calculated once at the beginning of the algorithm and be used for every CSI.

V. Application: GESS problem

Graph edit distance computation is extensively used in the solution of graph edit similarity search (GESS) problem. Given a collection 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 solution strategy 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 graphs using the expensive edit distance computations [6], [7], [8], [17], [18], [12], [19]. Upper bounds of GED could also be used to exempt some valid candidates from the expensive graph edit computations in the verification phase [6]. GESS problem can benefit from the `CSI_GED` algorithm in two different ways: (1) as a verifier with any GESS filtering method; (2) as a stand-alone GESS query method.

Incorporating the edit distance threshold τ with `CSI_GED` can further optimize it as follows. First, the possible set P_l at every search tree level l could be refined by removing a target edge e' if the cost value $\mathcal{C}(e_l, e')$ is greater than τ, i.e. if $\mathcal{C}(e_l, e') > \tau$. Indeed, those edges would not be part of any optimal edge map to the answering graphs. Likewise, the combine set C_l can be further optimized by adding a new pruning condition based on τ: A valid target edge e' is removed from the combine set C_l of any partial CSI f if, in addition to the upper bound based pruning, the map cost $g(f)$ added to the edge matching cost is greater than τ, i.e. if $g(f) + emc(e_l \rightarrow e') > \tau$, or if the lookahead cost $g'(f)$ of the map 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 edge map to the answering graphs. Finally, obtaining a complete CSI f with edit cost $g(f)$ is less than or equal to τ halts 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 on the graph edit distance, and in this case, the comparing graphs are surely within a distance less than τ. For 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.

VI. Experimental Results

In this section, we present a comprehensive experimental study on `CSI_GED`. All experiments were performed on a 3 GHz Dual Core CPU with 4G memory running Linux. `CSI_GED` is implemented in standard 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 the algorithm. The real data graphs are known to be sparse while the synthetic ones are always dense.

1) **AIDS**[6]. It is a DTP AIDS Antiviral Screen chemical compound dataset, published by National Cancer Institute. It consists of 42, 687 chemical compounds, with an average of 46 vertices and 48 edges. Compounds are labelled 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**[7]. It is a Program Dependence Graph (PDG) dataset

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

[7] www.comp.nus.edu.sg/∼xiaoli10/data/segos/linux_segos.zip

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 labelled 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) **PubChem**[8]. It is a chemical compound dataset. Chem_1M is a subset of PubChem, and consists of one million graphs. Chem_1M 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**[9]. It is a protein 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**. The synthetic datasets are generated using the synthetic graph data generator GraphGen[10]. The generator creates a collection of labeled, undirected and connected graphs. It allows us to specify various parameters such as dataset size, the average graph density, graph size, and the number of distinct vertex labels. For example, Syn10K.E30.D10.L5 means that it contains 10000 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, very small subsets of the AIDS dataset were used for testing the GESS query methods in [6], [7], [8], [17], [18], [12], [19], whereas the entire Linux dataset was only used for testing the filtering power of the approaches in [18], [20]. Chem_1M was recently used for testing graph edit similarity joins in cloud [21]. To carry out a comparative study on our machine, we chose 10K data graphs from AIDS, 100K from Chem_1M, and put 10000 seconds time limit for each algorithm to run. Besides this setting, we ran our algorithm on the entire real datasets and the results are shown in the scalability study.

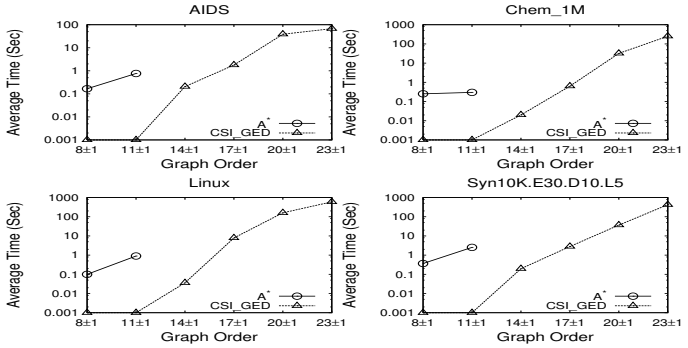

Fig. 10. Performance comparison with A^* algorithm against graph order.

[8]http://pubchem.ncbi.nlm.nih.gov

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

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

A. Evaluation Against Graph Order

In this set of experiments we compare the performance of CSI_GED with A^* algorithm against the graph order to show how large the graph order will be for each algorithm to work with on our machine. Six groups of data graphs were randomly selected from each dataset. Each group consists of three data graphs having three consecutive order values, and the number of vertices of each group is in the range: $8 \pm 1, 11 \pm 1, 14 \pm 1, 17 \pm 1, 20 \pm 1$, and 23 ± 1, resp. The executable of A^* was obtained from [11].

Figure 10 plots the average running time taken by each algorithm on each group, where graphs in the group are compared with each other in a self join manner. The figure shows A^* failed to run on groups consisting of large graphs, i.e. graphs with orders well beyond 12 vertices, for all datasets. This failure is due to the lack of memory–the physical memory available (4 GB) is not enough to store the huge number of partial vertex maps needed by A^*. In contrast, the very low memory requirements let CSI_GED run in any computational environment. Moreover, for the graph orders where A^* can run (7 and 12 vertices), CSI_GED significantly outperforms A^*. The performance gap starts small (from two to three orders of magnitude) and increases with graph order to become over three orders of magnitude. Thus, we conclude that CSI_GED is highly efficient for computing the edit distance on small graphs, and scales gracefully to large graphs.

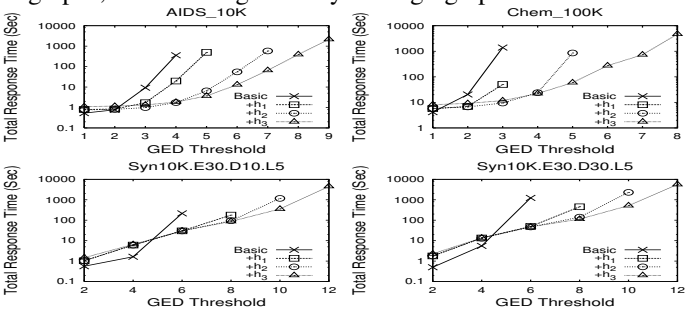

Fig. 11. Effect of the different heuristics on the performance of CSI_GED.

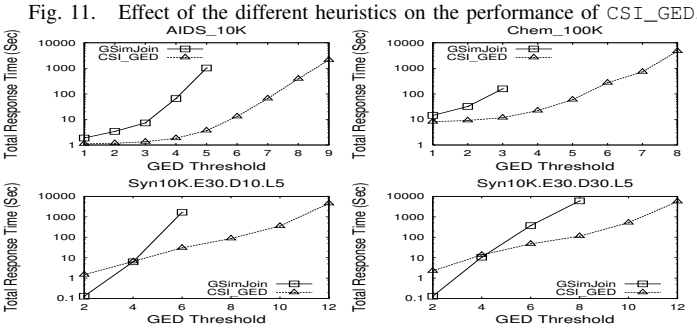

Fig. 12. Performance comparison with GSimJoin against edit threshold.

B. Effect of Heuristics

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 algorithm of Basic by incorporating the first heuristic (Section IV-A). "+h_2" denotes the improved algorithm of +h_1 by incorporating the second heuristic (Section IV-B). "+h_3" denotes the improved algorithm of +h_2 by incorporating the third heuristic (Section IV-C).

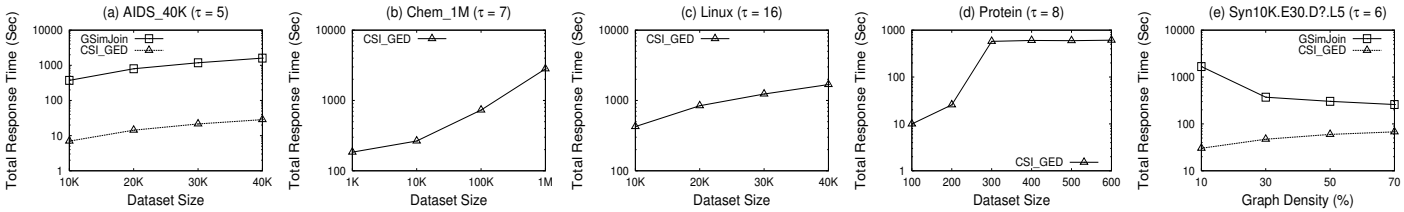

Fig. 13. Scalability on (1) dataset cardinality [a-d]; and (2) graph density [e].

We carried out these experiments in the context of graph edit similarity search. Figure 11 plots the time taken by each algorithm version on the different datasets at different τ. The figure shows 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 also shows that each heuristic enabled our algorithm to finish at larger thresholds 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 $\tau = 9$. And, the time taken by $+h_3$ at $\tau = 8$ is almost the same as that of `Basic` at $\tau = 4$.

The speedup achieved by each heuristic is clear from the figure. $+h_3$ brought around 120x 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 thresholds, the performance gap becomes clear at larger thresholds, e.g. at $\tau \geq 4$ on real datasets and at $\tau \geq 6$ on synthetic datasets. This performance boost from `basic` to $+h_3$ is attributed to the effective and less computationally-demanding pruning strategies.

C. Evaluation as a GESS Query Method

The third set of experiments is to 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` [12], [8] on real and synthetic graphs by varying the edit threshold τ.[11] `GSimJoin` is a path-based q-gram approach [12], [8]. It filters data graphs based on a counting condition on the number of matching q-grams with those of the query, as well as a global lower bound on graphs' labels. `GSimJoin` uses indexing for accelerating 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, `GSimJoin` performance is influenced by the gram size. For the real datasets, the best performance was obtained when $q = 4$, and when $q = 1$ for the synthetic ones. This variance is attributed to the fact that the graph density influences the number of path-based q-grams. The greater the graph density, the more path-based q-gram in a graph.

Figure 12 shows the effect of increasing the edit threshold on algorithms' performance. It reports the total response time taken by 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. These results are expected because `CSI_GED` implicitly uses lower and upper bounds as well as effective search order and pruning strategies to quickly confine the search space.

D. 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. Figure 13(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 (Figure 13(a)), where `GSimJoin` can run, the two methods are not very sensitive to this parameter, and the running time grows slowly. `CSI_GED` shows the best performance. The performance gain is consistent with the previous experiments. Figure 13(b)-(d) show that `CSI_GED` scales gracefully on other datasets.

Figure 13(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 improvement is brought to `GSimJoin` by A^*. Since the size of synthetically-generated graphs is fixed to 30 edges each, the graph order decreases with increasing density; it is about 6 vertices at higher densities. A^* is very efficient on comparing small and dense graphs.

VII. Related Work

H. Bunk [22] was the first to connect the graph edit distance problem with the one of maximum common subgraphs. In [23], Brun et al. 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. [24] exploited this relationship in the unit cost model, and derived lower and upper bounds of graph edit distance in the uncertain graph context. Different from [23] and [24], in this paper, we re-discovered the same relationship and utilized it in the development of a new efficient graph edit distance

[11] Other graph edit similarity search query methods do exist, such as `SEGOS` [18], `Pars` [17] and `Mixed` [19]. Experimental results in [17], [12], [8] 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 dataset. The current experimental results show that `CSI_GED` is 330x faster than `GSimJoin` on AIDS dataset at $\tau = 5$. Even though the executable of `Mixed` [19] was obtained from their authors, it is excluded from comparisons because it uses an approximated GED verifier and therefore its final results are not precise.

computation algorithm. To the best of our knowledge, we are the first to utilize this relationship to develop a new approach for GED computation.

The heart of our approach is to enumerate the common sub-structures that quickly lead to optimal edit paths. Even-though there exist many algorithms in the literature that can enumerate the common sub-structures [15], [3], [16], and some of them even employ backtracking search, none of these methods could serve our setting. First, because they are all vertex-based methods and adapting them would lead to algorithms suffering from problems similar to that of A^*-based methods. Second they could not be equipped with efficient pruning tools. In contrast, our approach is an edge-based backtracking search, which leverage the edit cost and accommodate new efficient pruning techniques.

Few approaches have been introduced to improve A^*, especially when using it as a verifier in the filter-and-verify approach for the GESS problem. [17] enhances A^* by starting its computations from an indexed common subgraph isomorphism. A^* then extends this isomorphism and uses τ to minimize memory, where extensions whose edit distance no more than τ are the only ones to be maintained. The main drawback is when there are more than one occurrence of the matching subgraph. In this case A^* should be run starting from every occurrence and it is not easy to share the computation among the different runs. To avoid this case, A^* starts from scratch. [12], [8] introduced another approach to enhance A^* based on the indexed, path-based q-grams. In this approach, A^* does not start with the matching parts (matching q-grams) as in the previous approach, it instead starts with the mismatching ones, because these q-grams incur some edit operations which helps terminating A^* very quickly on candidate graphs whose edit distance is not within τ. Mismatching q-grams are also used to enhance the search order in A^*, where the vertices contained by at least one mismatching q-gram 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 spanning tree. Using such order leverages the connectivity of a graph and can quickly find edge edit operations. Finally, $h(f)$ is optimized by using the maximum of global and local label lower bounds on the two remaining parts. All these improvements and much more are implicitly taken by our approach.

VIII. Conclusions

For decades, the widely used graph edit distance computation methods follow the best first search paradigm known as A^*. These methods have shown inability to compare large and distant graphs. To overcome the challenge of large graph comparisons, this paper introduced a novel approach for exact GED computation, called `CSI_GED`. `CSI_GED` utilizes backtracking combined with efficient heuristics to quickly search the edge-mapping space for those maps inducing optimal edit paths. Experiments showed that `CSI_GED` is highly efficient for computing the edit distance on small as well as large and distant graphs. Furthermore, `CSI_GED` is evaluated as a standalone graph edit similarity search query method. 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] A. Robles-Kelly and E. Hancock, "Graph edit distance from spectral seriation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 27(3), pp. 365–378, 2005.

[2] M. Neuhaus and H. Bunke, "Edit distance-based kernel functions for structural pattern classification," *Pattern Recognition*, vol. 39, pp. 1852–1863, 2006.

[3] J. Raymond, E. Gardiner, and P. Willett, "Rascal: Calculation of graph similarity using maximum common edge subgraphs," *The Computer J.*, vol. 45(6), pp. 631–644, 2002.

[4] D. Conte, P. Foggia, C. Sansone, and M. Vento, "Thirty years of graph matching in pattern recognition," *Int. J. Pattern Recognit. Artif. Intell.*, vol. 18, pp. 265–298, 2004.

[5] H. He and A. Singh, "Closure-tree: An index structure for graph queries," in *ICDE*, 2006, pp. 38–49.

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

[7] G. Wang, B. Wang, X. Yang, and G. Yu., "Efficiently indexing large sparse graphs for similarity search," *TKDE*, vol. 24(3), pp. 440–451, 2012.

[8] X. Zhao, C. Xiao, X. Lin, and W. Wang, "Efficient graph similarity joins with edit distance constraints," in *ICDE*, 2012, pp. 834–845.

[9] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. SSC*, vol. 4(2), pp. 100–107, 1968.

[10] K. Riesen, S. Fankhauser, and H. Bunke, "Speeding up graph edit distance computation with a bipartite heuristic," in *MLG*, 2007, pp. 21–24.

[11] K. Riesen, S. Emmenegger, and H. Bunke, "A novel software toolkit for graph edit distance computation," in *GbRPR*, 2013, pp. 142–151.

[12] X. Zhao, C. Xiao, X. Lin, W. Wang, and Y. Ishikawa, "Efficient processing of graph similarity queries with edit distance constraints," *VLDB J.*, vol. 22, pp. 727–752, 2013.

[13] S. EE, F. SH, and S. DA, "A network view of disease and compound screening," *Nat. Rev. Drug Discov.*, vol. 8(4), pp. 286–295, 2009.

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

[15] E. Krissinel and K. Henrick, "Common subgraph isomorphism detection by backtracking," *Software-Practice and Experience*, vol. 34, pp. 591–607, 2004.

[16] F. Abu-Khzam, N. Samatova, M. Rizk, and M. Langston, "The maximum common subgraph problem: faster solutions via vertex cover," in *AICCSA*, 2007, pp. 367–373.

[17] X. Zhao, C. Xiao, X. Lin, Q. Liu, and W. Zhang, "A partition based approach to structure similarity search," *PVLDB*, vol. 7(3), pp. 25–36, 2013.

[18] X. Wang, X. Ding, A. K. H. Tung, S. Ying, and H. Jin, "An efficient graph indexing method," in *ICDE*, 2012, pp. 210–221.

[19] W. Zheng, L. Zou, X. Lian, D. Wang, and D. Zhao, "Efficient graph similarity search over large graph databases," *TKDE*, vol. 27(4), pp. 964–978, 2015.

[20] K. Gouda and M. Arafa, "An improved global lower bound for graph edit similarity search," *Pattern Recognition Letters*, vol. 58, pp. 8–14, 2015.

[21] Y. Chen, X. Zhao, C. Xiao, W. Zhang, and J. Tang, "Efficient and scalable graph similarity joins in mapreduce," *The Scientific World Journal*, vol. 2014 (2014):749028. PMC., Web. 14 May 2015.

[22] H. Bunke, "On a relation between graph edit distance and maximum common subgraph," *Pattern Recognition Letters*, vol. 18(8), pp. 689–694, 1997.

[23] L. Brun, B. B. Gaüzère, and S. Fourey, "Relationships between graph edit distance and maximal common structural subgraph," *Technical Report (GREYC), http://hal.archives-ouvertes.fr/hal-00714879*, vol. 2012.

[24] W. Zheng, L. Zou, X. Lian, J. Yu, S. Song, and D. Zhao, "How to build templates for rdf question/answering an uncertain graph similarity join approach," in *SIGMOD*, 2015.