An Efficient Algorithm for Graph Edit Distance Computation

Xiaoyang Chen[a], Hongwei Huo[a,*], Jun Huan[b], Jeffrey Scott Vitter[c]

[a]*School of Computer Science and Technology, Xidian University, Xi'an 710071, China*
[b]*Baidu Research, Beijing 100094, China*
[c]*Department of Computer & Information Science, The University of Mississippi, University, MS 38677-1848, USA*

Abstract

The graph edit distance (GED) is a well-established distance measure widely used in many applications, such as bioinformatics, data mining, pattern recognition, and graph classification. However, existing solutions for computing the GED suffer from several drawbacks: large search spaces, excessive memory requirements, and many expensive backtracking calls. In this paper, we present BSS_GED, a novel vertex-based mapping method that calculates the GED in a reduced search space created by identifying invalid and redundant mappings. BSS_GED employs the beam-stack search paradigm, a widely utilized search algorithm in AI, combined with two specially designed heuristics to improve the GED computation, achieving a trade-off between memory utilization and expensive backtracking calls. Through extensive experiments, we demonstrate that BSS_GED is highly efficient on both sparse and dense graphs and outperforms the state-of-the-art methods. Furthermore, we apply BSS_GED to solve the well-investigated graph similarity search problem. The experimental results show that this method is dozens of times faster than state-of-the-art graph similarity search methods.

Keywords: Graph Edit Distance, Reduced Search Space, Beam-stack Search, Heuristics, Graph Similarity Search

1. Introduction

Graphs are widely used to model various structured data, including road maps [2], social networks [4], and molecular structures [8, 18]. Due to the extensive applications of graph data, considerable efforts have been made to develop techniques for effective graph data management and analysis, such as graph mining [5, 11], graph matching [6], and graph similarity search [24, 28].

*Corresponding author
Email addresses: `xychen1991@stu.mail.xidian.edu.cn` (Xiaoyang Chen), `hwhuo@mail.xidian.edu.cn` (Hongwei Huo), `huanjun@baidu.com` (Jun Huan), `jsv@OleMiss.edu` (Jeffrey Scott Vitter)

Within these studies, computing the similarity between two labeled graphs is a core and essential problem. In this paper, we focus on a similarity measure based on the graph edit distance (GED) because it can be applied to all types of graphs and can precisely capture the structural differences between graphs. Because of GED's flexible and error-tolerant characteristics, it has been successfully used in many fields, such as protein network analysis in bioinformatics [3], molecular comparison in chemistry [18], object recognition in computer vision [6], and graph classification [8, 10].

The GED of two graphs is defined as the minimum cost of an edit path between them, where an *edit path* is a sequence of edit operations (inserting, deleting, and relabeling vertices or edges) that transforms one graph into another. Unfortunately, computing the GED is known to be an NP-hard [27] problem. The GED's fault tolerance allows a vertex of one graph to be mapped to any vertex of the other graph, regardless of their labels and degrees. As a result, the complexity of computing the GED is exponential with respect to the number of vertices of the compared graphs.

The solutions for the GED computation are usually based on the tree-based search algorithm that explores all possible mappings of the vertices and edges of the compared graphs. This search space can be organized as an ordered search tree, where the inner nodes denote partial mappings and the leaf nodes denote complete mappings. Based on the way they generate node's successors, existing methods can be divided into two broad categories: *vertex-based* and *edge-based* mapping methods. When generating successors, the former method extends unmapped vertices of the compared graphs, while the latter method extends unmapped edges. A*-GED [17] and DF-GED [26] are two vertex-based mapping methods. A*-GED utilizes the best-first search paradigm A* [15] to extend partial mappings. The first complete mapping found yields the GED. During this process, however, A*-GED stores numerous partial mappings, leading to high memory consumption. To overcome this bottleneck, DF-GED adopts the depth-first search paradigm, requiring minimal memory. DF-GED also uses a branch-and-bound strategy to prune the useless search space. In contrast to A*-GED and DF-GED, CSI_GED [12] is a novel edge-based mapping method based on *common substructure isomorphism* and works well for sparse graphs. Similar to DF-GED, CSI_GED also adopts the depth-first search paradigm.

Although the above methods have achieved promising preliminary results, they still suffer from several drawbacks: (1) Both A*-GED and DF-GED attempt to enumerate all possible mappings between the compared graphs. However, some of these mappings are certainly unable to induce the minimum edit cost, called *invalid mappings*; and some induce the same edit cost, called *redundant mappings*. A better approach would be to avoid generating invalid mappings and to generate only one of the redundant mappings rather than many of them. Nevertheless, both A*-GED and DF-GED require a large search space because they generate numerous invalid and redundant mappings. (2) Although both DF-GED and CSI_GED perform a depth-first search to reduce the memory consumption, they easily become trapped into a local suboptimal solution and thus result in many expensive backtracking calls. (3) CSI_GED's search space is exponential with respect to the number of edges of the compared graphs, which makes CSI_GED unsuitable for dense graphs.

To solve the above issues, we propose a novel vertex-based mapping method, BSS_GED, based on the *beam-stack search* [20], which has demonstrated excellent performance in the AI literature. Our paper makes the following contributions:

- We propose a method of identifying invalid and redundant mappings and thereby can compute the GED in a reduced search space.
- We introduce the beam-stack search paradigm to establish a trade-off between memory utilization and backtracking calls during the GED computation, leading to better performance than the best-first and depth-first search paradigms. In addition, we propose two efficient heuristics to accelerate the search.
- Extensive experiments on both real and synthetic datasets show that BSS_GED is highly efficient on sparse as well as dense graphs and outperforms the state-of-the-art methods.
- We apply BSS_GED to solve the graph similarity search problem. The experimental results show that it is dozens of times faster than the state-of-the-art graph similarity search methods.

The remainder of this paper is organized as follows: In Section 2, we introduce the problem definition and summarize the vertex-based methods for the GED computation. In Section 3, we describe a method to identify invalid and redundant mappings. In Section 4, we show how to use the beam-stack search paradigm to compute the GED. In Section 5, we propose two heuristics to accelerate the search. In Section 6, we report the experimental results and analysis. We extend BSS_GED to create a standard graph similarity search query method in Section 7. Finally, we review related research works in Section 8 and draw conclusions in Section 9.

2. Preliminaries

In this section, we introduce some basic notation. For simplicity, we focus on simple undirected graphs without multi-edges or self-loops.

2.1. Problem Definition

Let Σ be a set of discrete-valued labels. A labeled graph is a triplet $G = (V_G, E_G, L)$, where V_G is the set of vertices, $E_G \subseteq V_G \times V_G$ is the set of edges, and $L : V_G \cup E_G \to \Sigma$ is a labeling function that assigns a label to a vertex or an edge. For a vertex u, we use $L(u)$ to denote its label. Similarly, $L(e(u, v))$ is the label of an edge $e(u, v)$. $\Sigma_{V_G} = \{L(u) : u \in V_G\}$ and $\Sigma_{E_G} = \{L(e(u, v)) : e(u, v) \in E_G\}$ are the label multisets of V_G and E_G, respectively. The graph size refers to $|V_G|$ in this paper.

Definition 1 (Subgraph Isomorphism [25])**.** *Given two graphs G and Q, G is subgraph isomorphic to Q, denoted by $G \subseteq Q$, if there exists an injective function $\phi : V_G \to V_Q$, such that (1) $\forall u \in V_G$, $\phi(u) \in V_Q$ and $L(u) = L(\phi(u))$, and (2) $\forall e(u, v) \in E_G$, $e(\phi(u), \phi(v)) \in E_Q$ and $L(e(u, v)) = L(e(\phi(u), \phi(v)))$. If $G \subseteq Q$ and $Q \subseteq G$, then G and Q are graph isomorphic to each other, denoted by $G \cong Q$.*

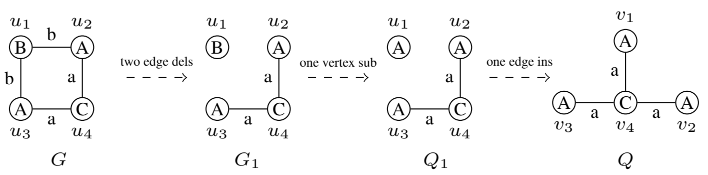

Figure 1: An optimal edit path P between graphs G and Q.

Given two graphs G and Q, six edit operations [22, 29] can be used to transform one graph into another: inserting/deleting an isolated vertex, inserting/deleting an edge, and substituting the label of a vertex or an edge. An *edit path* $P = \langle p_1, \ldots, p_k \rangle$ is a sequence of edit operations that transforms G to Q (or vice versa) such as $G = G^0 \xrightarrow{p_1}, \ldots, \xrightarrow{p_k} G^k \cong Q$. The edit cost of P is defined as the total cost of all the operations in P, i.e., $\sum_{i=1}^{k} c(p_i)$, where $c(p_i)$ is p_i's cost. In this paper, we focus on the uniform cost model, namely, $c(p_i) = 1$. Clearly, the edit cost of P is its length, denoted by $|P|$. We call P *optimal* only when it has the minimum length among all the possible edit paths.

Definition 2 (Graph Edit Distance)**.** *Given two graphs G and Q, the graph edit distance between them, denoted by $ged(G, Q)$, is the length of the optimal edit path that transforms G to Q (or vice versa).*

Example 1. Figure 1 shows an optimal edit path P between graphs G and Q. In this example, $|P| = 4$, where we delete two edges $e(u_1, u_2)$ and $e(u_1, u_3)$, substitute the label of vertex u_1 with label **A**, and insert one edge $e(u_1, u_4)$ with label **a**.

2.2. Graph Mapping

In this section, we introduce graph mapping between two graphs, which can induce an edit path between them. To match graphs G and Q of any size, we extend their vertex sets as follows: $V_G^* = V_G \cup \{\varepsilon\}$ and $V_Q^* = V_Q \cup \{\varepsilon\}$, where ε is a dummy vertex. We define the graph mapping as follows:

Definition 3 (Graph Mapping)**.** *A graph mapping from graph G to graph Q is a bijection $\psi : V_G^* \rightarrow V_Q^*$ such that $\forall u \in V_G^*$, $\psi(u) \in V_Q^*$, and at least one of u and $\psi(u)$ is not a dummy vertex.*

Given a graph mapping ψ from G to Q, it induces an unlabeled graph $H = (V_H, E_H)$, where $V_H = \{u : u \in V_G \wedge \psi(u) \in V_Q\}$ and $E_H = \{e(u, v) : e(u, v) \in E_G \wedge e(\psi(u), \psi(v)) \in E_Q\}$. Clearly, H is a common substructure of G and Q. Let G^ψ (resp., Q^ψ) be the labeled version of H embedded in G (resp., Q). We obtain an edit path $P_\psi : G \rightarrow G^\psi \rightarrow Q^\psi \rightarrow Q$ that transforms G to Q.

Let $C_D(\psi)$, $C_S(\psi)$, and $C_I(\psi)$ be the edit cost of transforming G to G^ψ, G^ψ to Q^ψ, and Q^ψ to Q, respectively. Because G^ψ is a subgraph of G, we only need to delete the vertices and edges in G that do not belong to G^ψ, when transforming G to G^ψ. Thus, $C_D(\psi) = |V_G| - |V_H| + |E_G| - |E_H|$. Similarly, $C_I(\psi) = |V_Q| - |V_H| + |E_Q| - |E_H|$. Because G^ψ and Q^ψ have the same structure H, we only need to substitute the corresponding vertex and edge labels between them. Thus, $C_S(\psi) = |\{u : u \in V_H \wedge L(u) \neq L(\psi(u))\}| + |\{e(u, v) : e(u, v) \in E_H \wedge L(e(u, v)) \neq L(e(\psi(u), \psi(v)))\}|$.

Theorem 1 ([12]). *Given a graph mapping ψ from graph G to graph Q, let P_ψ be the edit path induced by ψ. Then $|P_\psi| = C_D(\psi) + C_I(\psi) + C_S(\psi)$.*

Example 2. Consider the graphs G and Q in Figure 1. Given a graph mapping $\psi : \{u_1, u_2, u_3, u_4\} \rightarrow \{v_1, v_2, v_3, v_4\}$, where $\psi(u_1) = v_1$, $\psi(u_2) = v_2$, $\psi(u_3) = v_3$, and $\psi(u_4) = v_4$, we have $H = (\{u_1, u_2, u_3, u_4\}, \{e(u_2, u_4), e(u_3, u_4)\})$. Then ψ induces an edit path $P_\psi : G \rightarrow G^\psi \rightarrow Q^\psi \rightarrow Q$ shown in Figure 1, where $G^\psi = G_1$ and $Q^\psi = Q_1$. We compute that $C_D(\psi) = 2$, $C_I(\psi) = 1$, and $C_S(\psi) = 1$. From Theorem 1, the edit cost of P is $|P_\psi| = C_D(\psi) + C_I(\psi) + C_S(\psi) = 4$.

Among all possible graph mappings, we call the mapping that induces the optimal edit path *optimal*. Hereafter, for ease of presentation, we assume that G and Q are the two compared graphs and that $V_G = \{u_1, \ldots, u_{|V_G|}\}$ and $V_Q = \{v_1, \ldots, v_{|V_Q|}\}$. Given a processing order $\pi = [u_{i_1}, \ldots, u_{i_{|V_G|}}]$ of vertices in G, we rewrite the graph mapping ψ from $V_G^\star$ to $V_Q^\star$ as $\psi = \bigcup_{l=1}^{|V_G^\star|}\{(u_{i_l} \rightarrow v_{j_l})\}$ such that (1) $u_{i_l} = \varepsilon$ when $i_l > |V_G|$; (2) $v_{j_l} = \varepsilon$ when $j_l > |V_Q|$; and (3) $v_{j_l} = \psi(u_{i_l})$, for $1 \leq l \leq |V_G^*|$. Next we give an overview of the vertex-based mapping method of computing $ged(G, Q)$.

2.3. GED computation: Vertex-based Mapping Approach

Computing the GED of graphs G and Q is typically based on a tree-based search procedure that explores all possible graph mappings from G to Q. This search space can be organized as an ordered search tree, where the inner nodes denote partial graph mappings and the leaf nodes denote complete graph mappings. Such a search tree is created dynamically at runtime by iteratively generating successors linked by edges to the currently considered node.

Algorithm 1: BasicGenSuccr(r, l, π)

```
1  $succ \leftarrow \emptyset$;
2  $C_G^r \leftarrow V_G \backslash \{u_{i_1}, \ldots, u_{i_l}\}, C_Q^r \leftarrow V_Q \backslash \{v_{j_1}, \ldots, v_{j_l}\}$;
3  if $|C_G^r| > 0$ then
4      $u_{i_{l+1}} \leftarrow \pi[l+1]$;
5      foreach $z \in C_Q^r$ do
6          generate a successor $q$ such that $q \leftarrow r \cup \{(u_{i_{l+1}} \rightarrow z)\}$;
7          $succ \leftarrow succ \cup \{q\}$;
8      generate a successor $q$ such that $q \leftarrow r \cup \{(u_{i_{l+1}} \rightarrow \varepsilon)\}$;
9      $succ \leftarrow succ \cup \{q\}$;
10 else
11     generate a leaf node $\psi$ such that $\psi \leftarrow r \cup \bigcup_{z \in C_Q^r} \{(\varepsilon \rightarrow z)\}$;
12     $succ \leftarrow succ \cup \{\psi\}$;
13 return $succ$;
```

Consider a node $r = \{(u_{i_1} \rightarrow v_{j_1}), \ldots, (u_{i_l} \rightarrow v_{j_l})\}$ in layer l in the search tree, where $u_{i_k} (= \pi[k])$ is the kth processed vertex in G and v_{j_k} is the mapped vertex of u_{i_k}, for $1 \leq k \leq l$. Clearly, the sets of unmapped vertices in G and Q are $C_G^r = V_G \backslash \{u_{i_1}, \ldots, u_{i_l}\}$ and $C_Q^r = V_Q \backslash \{v_{j_1}, \ldots, v_{j_k}\}$, respectively. If $|C_G^r| > 0$, for the vertex $u_{i_{l+1}} = \pi[l + 1]$ to be processed, we choose a vertex z from C_Q^r or $\{\varepsilon\}$ as its mapped vertex and obtain a successor q of r such that $q = r \cup \{(u_{i_{l+1}} \to z)\}$. Otherwise, all the vertices in G have been processed; thus, we insert all the vertices in C_Q^r into G and obtain a leaf node $\psi = r \cup \bigcup_{z \in C_Q^r} \{(\varepsilon \to z)\}$. Algorithm 1 describes the process of generating r's successors.

Starting from a dummy node, $root = \emptyset$, we logically create the search tree layer by layer by iteratively generating successors using BasicGenSuccr. For a leaf node ψ, we compute its induced cost $|P_\psi|$ according to Theorem 1. Thus, when we generate all the leaf nodes, we must find an optimal graph mapping, and finally, we obtain $ged(G, Q)$. The existing methods A*-GED [17] and DF-GED [26] use the best-first and depth-first search paradigms to traverse this search tree to seek the optimal graph mapping, respectively.

Example 3. Consider the graphs G and Q in Figure 1. Figure 2 shows the entire search tree created by generating successors layer by layer using BasicGenSuccr, where the vertices in G are processed in the order $\pi = [u_1, u_2, u_4, u_3]$. The sequence of vertices on the path from the root to each leaf node yields a complete graph mapping. In this example, we find that the mapping $\psi = \{(u_1 \to v_1), (u_2 \to v_2), (u_4 \to v_4), (u_3 \to v_3)\}$ is optimal; therefore, $ged(G, Q) = |P_\psi| = 4$.

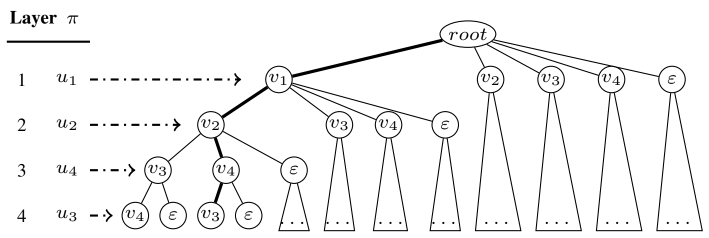

Figure 2: The entire search tree created by BasicGenSuccr.

However, the method BasicGenSuccr used in A*-GED [17] and DF-GED [26] generates all possible successors of each node. Consequently, both A*-GED and DF-GED may enumerate all possible graph mappings. Among these mappings, some are *invalid* when they are certainly unable to induce the optimal path; and some are *redundant* when they induce the same edit cost. We do not need to generate invalid mappings, and for redundant mappings, we only generate one version. To accomplish this, we next present how to identify invalid and redundant mappings.

3. Creating Reduced Search Space

3.1. Identifying Invalid Mappings

Let $|\psi|$ be the length of a graph mapping ψ. We estimate $|\psi|$ in Theorem 2, which can be used to identify invalid mappings.

Theorem 2. *Given an optimal graph mapping ψ from graph G to graph Q,* $|\psi| = \max\{|V_G|, |V_Q|\}$.

Proof. Suppose for the purpose of contradiction that $|\psi| > \max\{|V_G|, |V_Q|\}$. Then $(x \to \varepsilon)$ and $(\varepsilon \to y)$ must be present simultaneously in ψ, where $x \in V_G$ and $y \in V_Q$.

We construct another graph mapping $\psi' = (\psi \backslash \{(x \to \varepsilon), (\varepsilon \to y)\}) \cup \{(x \to y)\}$ and then prove that $|P_{\psi'}| < |P_\psi|$ as follows.

Let H and H' be two unlabeled graphs induced by ψ and ψ', respectively. Then $V_{H'} = \{u : u \in V_G \wedge \psi'(u) \in V_Q\} = \{u : u \in V_G \wedge \psi(u) \in V_Q\} \cup \{x : \psi'(x) \in V_Q\} = V_H \cup \{x\}$. Let $A_x = \{z : z \in V_H \wedge e(x,z) \in E_G \wedge e(y, \psi(z)) \in E_Q\}$. Then $E_{H'} = \{e(u,v) : e(u,v) \in E_G \wedge e(\psi'(u), \psi'(v)) \in E_Q\} = \{e(u,v) : e(u,v) \in E_G \wedge e(\psi(u), \psi(v)) \in E_Q\} \cup \{e(x,z) : z \in V_{H'} \wedge e(x,z) \in E_G \wedge e(y, \psi(z)) \in E_Q\} = E_H \cup \{e(x,z) : z \in A_x\}$. Because $x \notin V_H$, we have $e(x,z) \notin E_H$ for $z \in A_x$. Thus, $|V_{H'}| = |V_H| + 1$ and $|E_{H'}| = |E_H| + |A_x|$.

Because $C_D(\psi) = |V_G| - |V_H| + |E_G| - |E_H|$ and $C_I(\psi) = |V_Q| - |V_H| + |E_Q| - |E_H|$, we have $C_D(\psi') = C_D(\psi) - (1 + |A_x|)$ and $C_I(\psi') = C_I(\psi) - (1 + |A_x|)$. Because $C_S(\psi) = |\{u : u \in V_H \wedge L(u) \neq L(\psi(u))\}| + |\{e(u,v) : e(u,v) \in E_H \wedge L(e(u,v)) \neq L(e(\psi(u), \psi(v)))\}|$, we have $C_S(\psi') = C_S(\psi) + c(x \to y) + \sum_{z \in A_x} c(e(x,z) \to e(y, \psi(z)))$, where $c(\cdot)$ is the cost of relabeling a vertex or an edge such that $c(a \to b) = 0$ when $L(a) = L(b)$ and $c(a \to b) = 1$ otherwise. Thus, $C_S(\psi') \leq C_S(\psi) + 1 + |A_x|$. Therefore, $|P_{\psi'}| = C_D(\psi') + C_I(\psi') + C_S(\psi') \leq C_D(\psi) - (1 + |A_x|) + C_I(\psi) - (1 + |A_x|) + C_S(\psi) + 1 + |A_x| = |P_\psi| - (1 + |A_x|) < |P_\psi|$. This would be contradict the assertion that ψ is optimal. Hence, $|\psi| = \max\{|V_G|, |V_Q|\}$. $\square$

Theorem 2 states that a graph mapping must be invalid when its length is greater than $\max\{|V_G|, |V_Q|\}$. For example, consider the graphs G and Q in Figure 1. Given a graph mapping $\psi = \{(u_1 \to \varepsilon), (u_2 \to v_1), (u_4 \to v_3), (u_3 \to v_2), (\varepsilon \to v_3)\}$, we know that ψ with an edit cost of 5 must be invalid because $|\psi| = 5 > \max\{|V_G|, |V_Q|\} = 4$.

3.2. Identifying Redundant Mappings

For a vertex u in V_Q, its neighborhood information is defined as $N_Q(u) = \{(v, L(e(u,v))) : v \in V_Q \wedge e(u,v) \in E_Q\}$.

Definition 4 (Vertex Isomorphism)**.** *Given two vertices $u, v \in V_Q$, u is isomorphic to v, denoted by $u \sim v$, if and only if $L(u) = L(v)$ and $N_Q(u) = N_Q(v)$.*

By Definition 4, it is trivial to find that the isomorphic relationship between vertices is an equivalence relation. Thus, we can divide V_Q into λ_Q equivalent classes $V_Q^1, \ldots, V_Q^{\lambda_Q}$ of isomorphic vertices. Each vertex u is said to belong to class $\rho(u) = m$ if $u \in V_Q^m$. The dummy vertices in $\{\varepsilon\}$ are isomorphic to each other, and let $\rho(\varepsilon) = \lambda_Q + 1$.

Definition 5 (Canonical Code)**.** *Given a graph mapping $\psi = \bigcup_{l=1}^{|V_G^*|} \{(u_{i_l} \to v_{j_l})\}$, where $v_{j_l} = \psi(u_{i_l})$ for $1 \leq l \leq |V_G^*|$, the canonical code of ψ is defined as $code(\psi) = \langle \rho(v_{j_1}), \ldots, \rho(v_{j_{|\psi|}}) \rangle$.*

Given two graph mappings ψ and ψ' of the same length, we say that *code*$(\psi) =$ *code*(ψ') if and only if $\rho(v_{j_l}) = \rho(v'_{j_l})$ for $1 \leq l \leq |\psi|$, where $v_{j_l} = \psi(u_{i_l})$ and $v'_{j_l} = \psi'(u_{i_l})$.

Theorem 3. *Given two graph mappings ψ and ψ', let P_ψ and $P_{\psi'}$ be edit paths induced by ψ and ψ', respectively. If code(ψ) = code(ψ') then $|P_\psi| = |P_{\psi'}|$.*

Proof. As discussed in Section 2.2, $|P_\psi| = C_I(\psi) + C_D(\psi) + C_S(\psi)$. To prove that $|P_\psi| = |P_{\psi'}|$, we first prove that $C_I(\psi) = C_I(\psi')$ and $C_D(\psi) = C_D(\psi')$; then, we prove that $C_S(\psi) = C_S(\psi')$.

Let H and H' be two unlabeled graphs induced by ψ and ψ', respectively. For a vertex u in V_H, $\psi(u)$ is u's mapped vertex. Because *code*(ψ) = *code*(ψ'), we have $\rho(\psi(u)) = \rho(\psi'(u))$, and hence, we obtain $\psi(u) \sim \psi'(u)$. Because $\psi(u) \neq \varepsilon$, we know that $\psi'(u) \neq \varepsilon$ by Definition 4. Thus, $u \in V_{H'}$, and hence, we obtain $V_H \subseteq V_{H'}$. Similarly, we also obtain $V_{H'} \subseteq V_H$. Therefore, $V_H = V_{H'}$.

For an edge $e(u, v)$ in E_H, $e(\psi(u), \psi(v))$ is its mapped edge in E_Q. Because $\rho(\psi(u)) = \rho(\psi'(u))$, we have $\psi(u) \sim \psi'(u)$, and then, we obtain $N_Q(\psi(u)) = N_Q(\psi'(u))$. Thus, we have $e(\psi'(u), \psi(v)) \in E_Q$. Because $\rho(\psi(v)) = \rho(\psi'(v))$, we know that $\psi(v) \sim \psi'(v)$; consequently, edges must exist between $\psi(u)$ and $\psi'(v)$, and between $\psi'(u)$ and $\psi'(v)$ (an illustration is shown in Figure 3). Therefore, $e(\psi'(u), \psi'(v)) \in E_Q$, and hence, $e(u, v) \in E_{H'}$. Thus, we have $E_H \subseteq E_{H'}$. Similarly, we also obtain $E_{H'} \subseteq E_H$. Therefore, $E_H = E_{H'}$.

Because $V_H = V_{H'}$ and $E_H = E_{H'}$, we have $H = H'$. Thus, $C_I(\psi) = C_I(\psi')$ and $C_D(\psi) = C_D(\psi')$. Hereafter, we do not distinguish between H and H'.

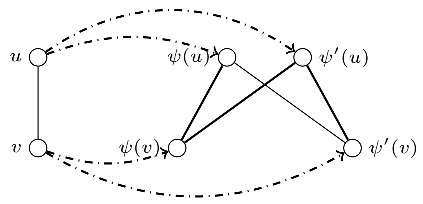

Figure 3: Illustration of isomorphic vertices.

For any vertex u in V_H, we have $L(\psi(u)) = L(\psi'(u))$ because $\psi(u) \sim \psi'(u)$. Thus, $|\{u : u \in V_H \wedge L(u) \neq L(\psi(u))\}| = |\{u : u \in V_H \wedge L(u) \neq L(\psi'(u))\}|$. For any edge $e(u, v)$ in E_H, because $\psi(u) \sim \psi'(u)$, we know that $L(e(\psi(u), \psi(v))) = L(e(\psi'(u), \psi(v)))$. Similarly, we obtain $L(e(\psi'(u), \psi(v))) = L(e(\psi'(u), \psi'(v)))$ because $\psi(v) \sim \psi'(v)$. Thus, $L(e(\psi(u), \psi(v))) = L(e(\psi'(u), \psi'(v)))$. Hence, we have $|\{e(u, v) : e(u, v) \in E_H \wedge L(e(u, v)) \neq L(e(\psi(u), \psi(v)))\}| = |\{e(u, v) : e(u, v) \in E_H \wedge L(e(u, v)) \neq L(e(\psi'(u), \psi'(v)))\}|$. Therefore, $C_S(\psi) = C_S(\psi')$, and consequently, $|P_\psi| = |P_{\psi'}|$. $\square$

Example 4. Consider the graphs G and Q shown in Figure 1. For Q, we know that $L(v_1) = L(v_2) = L(v_3) = \mathbf{A}$, and $N_Q(v_1) = N_Q(v_2) = N_Q(v_3) = \{(v_4, \mathbf{a})\}$; thus, $v_1 \sim v_2 \sim v_3$. Hence, we can divide V_Q into two equivalent classes: $V_Q^1 = \{v_1, v_2, v_3\}$ and $V_Q^2 = \{v_4\}$. Clearly, $\rho(v_1) = \rho(v_2) = \rho(v_3) = 1$ and $\rho(v_4) = 2$. Given two graph mappings, $\psi = \{(u_1 \to v_1), (u_2 \to v_2), (u_4 \to v_4), (u_3 \to v_3)\}$ and $\psi' = \{(u_1 \to v_2), (u_2 \to v_3), (u_4 \to v_4), (u_3 \to v_1)\}$, we know that *code*$(\psi) = \langle\rho(v_1), \rho(v_2), \rho(v_4), \rho(v_3)\rangle = \langle 1, 1, 2, 1\rangle$. Similarly, we compute that *code*$(\psi') = \langle 1, 1, 2, 1\rangle$. Thus, we have *code*$(\psi)$ = *code*(ψ'); therefore, $|P_\psi| = |P_{\psi'}| = 4$.

Theorem 3 states that graph mappings with the same canonical code induce the same edit cost. Thus, among those mappings, only one needs to be generated. Next, we incorporate Theorems 2 and 3 into BasicGenSuccr to avoid generating the invalid and redundant mappings discussed above.

3.3. Generating Successors

Consider a node $r = \{(u_{i_1} \to v_{j_1}), \ldots, (u_{i_l} \to v_{j_l})\}$ in the search tree. Then the sets of unmapped vertices in G and Q are $C_G^r = V_G \backslash \{u_{i_1}, \ldots, u_{i_l}\}$ and $C_Q^r = V_Q \backslash \{v_{j_1}, \ldots, v_{j_l}\}$, respectively. Let $z \in C_Q^r \cup \{\varepsilon\}$ be a possible mapped vertex of the vertex $u_{i_{l+1}}$ to be processed.

According to Theorem 2, if $|V_G| \leq |V_Q|$, then $|\psi| = |V_Q|$; this means that none of the vertices in V_G can be mapped to a dummy vertex, i.e., $(u \to \varepsilon) \notin \psi$ for $\forall u \in V_G$. Therefore, we establish Rule 1 as follows:

Rule 1. If $|C_G^r| \leq |C_Q^r|$, then $z \in C_Q^r$; otherwise, $z = \varepsilon$ or $z \in C_Q^r$.

Applying Rule 1 to BasicGenSuccr (i.e., Alg. 1) to generate the successors of each node, we know that when $|V_G| \leq |V_Q|$, none of the vertices in V_G are mapped to ε; otherwise, only $|V_G| - |V_Q|$ vertices are mapped as such. Consequently, the obtained graph mapping ψ must satisfy $|\psi| = \max\{|V_G|, |V_Q|\}$.

Definition 6 (Partial Order of Canonical Code). *Let ψ and ψ' be two graph mappings such that $code(\psi) = code(\psi')$. We define that $\psi \preceq \psi'$, if $\exists l, 1 \leq l \leq |\psi|$, satisfies $\psi(u_{i_k}) = \psi'(u_{i_k})$ for $1 \leq k < l$ and $\psi(u_{i_l}) < \psi'(u_{i_l})$.*

Among the graph mappings with the same canonical code, we only need to generate the smallest according to the partial order introduced in Definition 6. Specifically, we map $u_{i_{l+1}}$ to the smallest unmapped vertex in V_Q^m for $1 \leq m \leq \lambda_Q$. Therefore, we establish Rule 2 as follows:

Rule 2. $z \in \bigcup_{m=1}^{\lambda_Q} \min\{C_Q^r \cap V_Q^m\}$.

Based on Rules 1 and 2, we propose the method of generating r's successors in Algorithm 2. Lines 5–9 correspond to Rule 2, and lines 10–12 correspond to Rule 1. Note that we have divided V_Q into λ_Q equivalent classes $V_Q^1, \ldots, V_Q^{\lambda_Q}$ before this algorithm executes.

Example 5. Consider the graphs G and Q in Figure 1. Figure 4 shows the entire search tree created by iteratively generating successors using GenSuccr. In this example, we generate a total of four graph mappings and trivially compute that $ged(G, Q) = 4$. In contrast, the search tree created by BasicGenSuccr shown in Figure 2 contains more than 24 graph mappings.

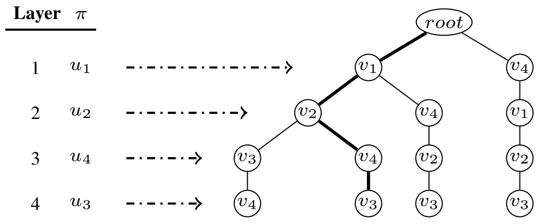

Figure 4: The entire search tree created by GenSuccr.

Algorithm 2: GenSuccr(r, l, π)

```
1  $succ \leftarrow \emptyset$;
2  $C_G^r \leftarrow V_G \backslash \{u_{i_1}, \ldots, u_{i_l}\}, C_Q^r \leftarrow V_Q \backslash \{v_{j_1}, \ldots, v_{j_l}\}$;
3  if $|C_G^r| > 0$ then
4      $u_{i_{l+1}} \leftarrow \pi[l+1]$;
5      for $m \leftarrow 1$ to $\lambda_Q$ do
6          if $C_Q^r \cap V_Q^m \neq \emptyset$ then
7              $z \leftarrow \min\{C_Q^r \cap V_Q^m\}$;
8              generate a successor $q$ such that $q \leftarrow r \cup \{(u_{i_{l+1}} \rightarrow z)\}$;
9              $succ \leftarrow succ \cup \{q\}$;
10     if $|C_G^r| > |C_Q^r|$ then
11         generate a successor $q$ such that $q \leftarrow r \cup \{(u_{i_{l+1}} \rightarrow \varepsilon)\}$;
12         $succ \leftarrow succ \cup \{q\}$;
13 else
14     generate a leaf node $\psi$ such that $\psi \leftarrow r \cup \bigcup_{z \in C_Q^r} \{(\varepsilon \rightarrow z)\}$;
15     $succ \leftarrow succ \cup \{\psi\}$;
16 return $succ$;
```

3.4. Search Space Analysis

By replacing BasicGenSuccr with GenSuccr to generate successors, we eliminate many invalid and redundant mappings, and thus we create a reduced search tree. In this section, we analyze the size of the search tree, denoted by $\mathcal{S}_R$, which is the total number of nodes in the search tree.

Nodes in the search tree are grouped into different layers based on their distances from the root node. Hence, the search tree is divided into layers, one layer for each depth. When all the vertices in V_G have been processed, for any node in layer $|V_G|$ (starting from 0), we generate a unique leaf node (see line 14 in GenSuccr); thus, we can regard this layer as the last layer. Specifically, we only need to generate the first $|V_G|$ layers. Let N_l be the number of nodes in layer l. Therefore, $\mathcal{S}_{\mathcal{R}} = \sum_{l=0}^{|V_G|} N_l$.

Let $B_G^l = \{u_{i_1}, \ldots, u_{i_l}\}$ be the set of processed vertices in G in layer l. Correspondingly, we choose l vertices from $V_Q \cup \{\varepsilon\}$ as their mapped vertices. Let $B_Q^l = \{v_{j_1}, \ldots, v_{j_l}\}$ be the l selected vertices. We use a vector $\mathbf{x} = [x_1, \ldots, x_{\lambda_Q+1}]$ to represent B_Q^l, where x_m is the number of vertices in B_Q^l that belong to V_Q^m, i.e., $x_m = |B_Q^l \cap V_Q^m|$ for $1 \leq m \leq \lambda_Q$, and x_{λ_Q+1} is the number of dummy vertices in B_Q^l. Thus, we have

$$\sum_{m=1}^{\lambda_Q+1} x_m = l, \tag{1}$$

where $0 \leq x_m \leq |V_Q^m|$ for $1 \leq m \leq \lambda_Q$, and $x_{\lambda_Q+1} \geq 0$.

A solution $\mathbf{x}$ to Equation (1) corresponds to a unique B_Q^l. The reason for this is as follows: In Rule 2, we select the smallest unmapped vertex in V_Q^m as the mapped vertex each time. Thus, for x_m in $\mathbf{x}$, B_Q^l will contain the first x_m smallest vertices in V_Q^m. For example, consider the search tree in Figure 4. Let $l = 3$ and $\mathbf{x} = [2, 1, 0]$. B_Q^l will contain the first two smallest vertices v_1 and v_2 in V_Q^1 and the smallest vertex v_4 in V_Q^2; thus, $B_Q^l = \{v_1, v_2, v_4\}$.

Let Ψ_l be the set of solutions for Equation (1). Then, Ψ_l covers all possible B_Q^l. For a solution $\mathbf{x} \in \Psi_l$, it produces a total of $l!/\prod_{m=1}^{\lambda_Q+1} x_m!$ different (partial) canonical codes. For example, for $\mathbf{x} = [2, 1, 0]$, we produce three partial canonical codes $\langle 1, 1, 2\rangle$, $\langle 1, 2, 1\rangle$ and $\langle 2, 1, 1\rangle$. We know that each (partial) canonical code corresponds to a (partial) mapping from B_G^l to B_Q^l; thus,

$$N_l = \sum_{\mathbf{x}\in\Psi_l} \frac{l!}{\prod_{m=1}^{\lambda_Q+1} x_m!}. \tag{2}$$

From Rule 1, we select ε as the mapped vertex only when there are more unmapped vertices in G than in Q. Consequently, the number of dummy vertices in $B_Q^{|V_G|}$ is 0 when $|V_G| \le |V_Q|$ and $|V_G| - |V_Q|$ otherwise. Let $l = |V_G|$. We consider the following two cases:

Case I. When $|V_G| > |V_Q|$, for any $\mathbf{x} \in \Psi_{|V_G|}$, we have $x_{\lambda_Q+1} = |V_G| - |V_Q|$. Equation (1) can be reduced to $\sum_{m=1}^{\lambda_Q} x_m = |V_Q|$. Because $\sum_{m=1}^{\lambda_Q} |V_Q^m| = |V_Q|$ and $0 \le x_m \le |V_Q^m|$ for $1 \le m \le \lambda_Q$, Equation (1) has a unique solution $\mathbf{x} = [|V_Q^1|, \ldots, |V_Q^{\lambda_Q}|, |V_G| - |V_Q|]$. Substituting $\mathbf{x}$ into Equation (2), we have

$$N_{|V_G|} = \frac{|V_G|!}{(|V_G| - |V_Q|)! \prod_{m=1}^{\lambda_Q} |V_Q^m|!}. \tag{3}$$

Because $N_0 = 1$ and $N_1 \le \cdots \le N_{|V_G|}$, we have

$$\mathcal{S}_{\mathcal{R}} \le |V_G| \frac{|V_G|!}{(|V_G| - |V_Q|)! \prod_{m=1}^{\lambda_Q} |V_Q^m|!} + 1. \tag{4}$$

Case II. When $|V_G| \le |V_Q|$, for any $\mathbf{x} \in \Psi_{|V_G|}$, we have $x_{\lambda_Q+1} = 0$. Equation (1) can be reduced to $\sum_{m=1}^{\lambda_Q} x_m = |V_G|$. Because $|V_G| \le |V_Q|$, we have

$$N_{|V_G|} = \sum_{\substack{\mathbf{x}\in\Psi_{|V_G|},\\ x_{\lambda_Q+1}=0}} \frac{|V_G|!}{\prod_{m=1}^{\lambda_Q} x_m!} \le \sum_{\substack{\mathbf{x}\in\Psi_{|V_Q|},\\ x_{\lambda_Q+1}=0}} \frac{|V_Q|!}{\prod_{m=1}^{\lambda_Q} x_m!} = \frac{|V_Q|!}{\prod_{m=1}^{\lambda_Q} |V_Q^m|!}. \tag{5}$$

Because $N_0 = 1$ and $N_1 \le \cdots \le N_{|V_G|}$, we have

$$\mathcal{S}_{\mathcal{R}} \le |V_G| \frac{|V_Q|!}{\prod_{m=1}^{\lambda_Q} |V_Q^m|!} + 1. \tag{6}$$

In Case II, when we do not consider the isomorphic vertices in B_Q^l, a total of $l!$ mappings from B_G^l to B_Q^l exist. Because there are at most $\binom{|V_Q|}{l}$ possible B_Q^l, we have

$$N_l \leq \binom{|V_Q|}{l} \cdot l! = \frac{|V_Q|!}{(|V_Q|-l)!}. \tag{7}$$

Therefore,

$$\begin{aligned}\mathcal{S}_R &= \sum_{l=0}^{|V_G|} N_l \leq \sum_{l=1}^{|V_G|} \frac{|V_Q|!}{(|V_Q|-l)!} + 1 \\ &= \frac{|V_Q|!}{(|V_Q|-|V_G|)!} \sum_{l=1}^{|V_G|} \frac{1}{\prod_{m=1}^{|V_G|-l}(|V_Q|-|V_G|+m)} + 1 \\ &\leq 2\frac{|V_Q|!}{(|V_Q|-|V_G|)!} + 1. \end{aligned} \tag{8}$$

In summary, by (4), (6), and (8), we have

$$\mathcal{S}_R = \begin{cases} \mathcal{O}\big(\frac{|V_G||V_G|!}{(|V_G|-|V_Q|)!\prod_{m=1}^{\lambda_Q}|V_Q^m|!}\big) & \text{if } |V_G| > |V_Q| \\ \mathcal{O}\big(\min\{\frac{|V_G||V_Q|!}{\prod_{m=1}^{\lambda_Q}|V_Q^m|!}, \frac{|V_Q|!}{(|V_Q|-|V_G|)!}\}\big) & \text{otherwise} \end{cases} \tag{9}$$

4. GED Computation using Beam-stack Search

The previous section showed how we created a reduced search space; however, we still need an efficient search paradigm to traverse this search space to seek an optimal graph mapping. In this section, by incorporating the *beam-stack search* [20], we give the approach to compute the GED.

4.1. Data Structures

For a node r in the search tree, $f(r) = g(r) + h(r)$ is the total edit cost (also called the f-cost) assigned to r, where $g(r)$ is the incurred edit cost from *root* to r, and $h(r)$ is the estimated edit cost from r to a leaf node that is less than or equal to the real cost. $g(r)$ and $h(r)$ will be discussed in Section 5.1. Before formally presenting the algorithm, we first introduce the data structures used.

- A beam stack, bs, is a generalized stack in which each item is a half-open interval $[f_{\min}, f_{\max})$. We use $bs[l]$ to denote the interval of layer l. For a node r in layer l, its successor n in the next layer $l+1$ is allowed to expand only when $f(n)$ lies in the interval $bs[l]$ (i.e., $bs[l].f_{\min} \leq f(n) < bs[l].f_{\max}$).
- Priority queues, $open[0], \ldots, open[|V_G|]$, where $open[l]$, for $0 \leq l \leq |V_G|$, stores the expanded nodes in layer l. We use $|open[l]|$ to denote the number of nodes in $open[l]$.
- A table, new, where $new[\mathcal{H}(r)]$ stores r's successors and $\mathcal{H}(r)$ is a hash function that assigns a unique ID to r.

4.2. Algorithm

Algorithm 3 performs an iterative search to obtain an increasingly tight upper bound ub until $ub = ged(G, Q)$. In each iteration, we perform the following two steps: (1) We first use beam search [19] to quickly find a leaf node whose cost is an upper bound of $ged(G, Q)$. Then we update ub (line 6). Because the beam search expands at most w nodes in each layer, some nodes are *inadmissibly pruned* (i.e., nodes are pruned due to memory limitations, which may cause the algorithm to miss the optimal solution) when the number of nodes in a layer is greater than the beam width w. (2) We backtrack and pop items from bs until bs.top().$f_{\max} < ub$ (lines 7–8). Let l be the layer where we stop backtracking. When $l = -1$, we have completed the search and obtain $ub = ged(G, Q)$ (lines 9–10); otherwise, we shift the range of bs.top() (line 11) to re-expand the inadmissibly pruned nodes from layer l and continue the search for a tighter ub in the next iteration. Note that at the beginning of this algorithm, we first determine the vertex processing order of G (line 1, see Section 5.2) and then divide the vertices in Q into λ_Q subsets of isomorphic vertices (line 2, see Section 3.2).

The procedure BeamSearch performs a beam search to seek a tighter ub, where PQL and $PQLL$ are two temporary priority queues used to record expanded nodes in two adjacent layers. In each layer, we sequentially pop nodes according to their f-cost[1]. Let r be the node currently associated with the smallest f-cost (line 4). If r is a leaf node, we update ub and stop the search because $g(z) \geq g(r)$ holds for $\forall z \in PQL$ (line 7); otherwise, we call ExpandNode to generate r's successors that are allowed to expand and insert them into $PQLL$ (lines 8–9). Because at most w successors are allowed to expand in each layer, we keep only the best w nodes (i.e., those with the smallest f-cost) in $PQLL$. The remaining nodes are inadmissibly pruned (lines 10–13). Correspondingly, we modify the right boundary of bs.top() as the lowest f-cost among all the inadmissibly pruned nodes (line 12). After that, we move to the next layer and repeat the above process until we find a leaf node (lines 14–15).

The ExpandNode procedure generates r's successors that are allowed to expand. When first generated, all nodes are marked as false. If r has not been visited (i.e., $r.visited =$ false), then we call GenSuccr to generate r's successors and mark r as visited (lines 2–4); otherwise, we read r's successors directly from new (line 6). For a successor n of r, if $f(n) \geq ub$ or $n.visited =$ true, then we can safely prune it (see Lemma 1); meanwhile, we delete its successors from new (line 9). Otherwise, when bs.top().$f_{\min} \leq f(n) < bs$.top().$f_{\max}$, we expand n. When all the successors of r have been safely pruned, we prune r and delete its successors from new (line 13).

Lemma 1. *In ExpandNode, if $f(n) \geq ub$ or $n.visited = true$, i.e., line 8, we can safely prune n.*

Proof. In ExpandNode, we assume that r and its successors n are in layers l and $l + 1$, respectively. For the case in which $f(n) \geq ub$, the proof is trivial. Next we prove this lemma in the other case.

[1]When two or more nodes have the same f-cost, we use the tie-breaking rule in [20] to impose a total ordering on these nodes.

Algorithm 3: BSS_GED(G, Q, w)

```
1  $\pi \leftarrow$ DetermineOrder$(G)$;
2  divide $V_Q$ into $\lambda_Q$ equivalent classes $V_Q^1, \dots, V_Q^{\lambda_Q}$;
3  $root \leftarrow \emptyset$, $bs \leftarrow \emptyset$, $open[] \leftarrow \emptyset$, $new[] \leftarrow \emptyset$, $l \leftarrow 0, ub \leftarrow \infty$;
4  $bs$.push$([0, ub])$, $open[0]$.push$(root)$;
5  while $bs \neq \emptyset$ do
6      BeamSearch $(l, ub, bs, open, new, \pi)$;
7      while $bs$.top$().f_{\max} \geq ub$ do
8          $bs$.pop$(), l \leftarrow l - 1$;
9      if $l = -1$ then
10         return $ub$;
11     $bs$.top$().f_{\min} \leftarrow bs$.top$().f_{\max}$, $bs$.top$().f_{\max} \leftarrow ub$;
12 return $ub$;
   procedure BeamSearch$(l, ub, bs, open, new, \pi)$
1      $PQL \leftarrow open[l]$, $PQLL \leftarrow \emptyset$;
2      while $PQL \neq \emptyset$ or $PQLL \neq \emptyset$ do
3          while $PQL \neq \emptyset$ do
4              $r \leftarrow \arg\min_n\{f(n) : n \in PQL\}$;
5              $PQL \leftarrow PQL \backslash \{r\}$;
6              if $r$ is a complete graph mapping then
7                  $ub \leftarrow \min\{ub, g(r)\}$, return;
8              $succ \leftarrow$ExpandNode $(r, l, ub, new, \pi)$;
9              $PQLL \leftarrow PQLL \cup succ$;
10         if $|PQLL| > w$ then
11             $keepNodes \leftarrow$ the best $w$ nodes in $PQLL$;
12             $bs$.top$().f_{\max} \leftarrow \min\{f(n) : n \in PQLL \wedge n \notin keepNodes\}$;
13             $PQLL \leftarrow keepNodes$;
14         $open[l+1] \leftarrow PQLL, PQL \leftarrow PQLL$;
15         $PQLL \leftarrow \emptyset, l \leftarrow l + 1, bs$.push$([0, ub])$;
   procedure ExpandNode$(r, l, ub, new, \pi)$
1      $expand \leftarrow \emptyset$;
2      if $r.visited =$ false then
3          $succ \leftarrow$GenSuccr$(r, l, \pi)$;
4          $new[\mathcal{H}(r)] \leftarrow succ, r.visited \leftarrow$ true;
5      else
6          $succ \leftarrow new[\mathcal{H}(r)]$;
7      foreach $n \in succ$ do
8          if $f(n) \geq ub$ or $n.visited =$ true then
9              $new[\mathcal{H}(n)] \leftarrow \emptyset$, continue;
10         if $bs$.top$().f_{\min} \leq f(n) < bs$.top$().f_{\max}$ then
11             $expand \leftarrow expand \cup \{n\}$;
12     if $\forall n \in succ, f(n) \geq ub$ or $n.visited =$ true then
13         $new[\mathcal{H}(r)] \leftarrow \emptyset$;
14     return $expand$;
```

Consider the items in bs in the last iteration. Assume that we perform the BeamSearch from layer k (i.e., we backtrack to layer k in the last iteration, see lines 7–8 in Alg. 3). Clearly, $k \leq l$ and $bs[m].f_{\max} \geq ub$ for $l+1 \leq m \leq |V_G|$. If $n.visited = \text{true}$, then we have called ExpandNode to generate n's successors in the last iteration. For a successor x of n in layer $l+2$, if x is inadmissibly pruned, then $f(x) \geq bs[l+1].f_{\max} \geq ub$; thus we can safely prune x. Otherwise, we consider a successor of x and repeat this decision process until we reach a leaf node z. Then the condition that $f(z) = g(z) \geq ub$ must hold. Therefore, none of n's descendants can produce a tighter ub; consequently, we can safely prune n. □

Lemma 2. *A node r is visited at most $\mathcal{O}(|V_Q|)$ times.*

Proof. For a node r in layer l, GenSuccr generates at most $|V_Q|+1$ successors; thus, at most $|open[l]| \cdot (|V_Q|+1)$ nodes are expanded in layer l in each iteration. To fully generate all the successors in layer $l+1$, we backtrack to layer l at most $|open[l]| \cdot (|V_Q|+1)/w \leq |V_Q|+1$ times because $|open[l]| \leq w$. Later, when we again visit r, all the successors of r have either been pruned or marked; thus, we can safely prune them according to Lemma 1. Consequently, r cannot produce a tighter ub during this iteration; thus we can safely prune it (i.e., lines 12–13 in ExpandNode). Adding the first time when generating r, we visit r at most $|V_Q|+3$ times–that is, $\mathcal{O}(|V_Q|)$. □

Theorem 4. *Given two graphs G and Q, BSS_GED returns $ged(G,Q)$ and its time complexity is $\mathcal{O}(|V_Q| \cdot \mathcal{S}_R)$, where $\mathcal{S}_R$ is the total number of nodes in the search tree.*

Proof. From Lemma 2, a node is visited at most $\mathcal{O}(|V_Q|)$ times; thus, all nodes are visited at most $\mathcal{O}(|V_Q| \cdot \mathcal{S}_R)$ times (see $\mathcal{S}_R$ in Section 3.4). Therefore, BSS_GED always terminates in a finite number of iterations and its time complexity is $\mathcal{O}(|V_Q| \cdot \mathcal{S}_R)$. In BeamSearch, we update $ub = \min\{ub, g(r)\}$ each time. Thus ub becomes progressively tighter. We prove by contradiction that ub converges to $ged(G,Q)$ when BSS_GED terminates.

Suppose that $ub > ged(G,Q)$. Let x and y be two leaf nodes such that $g(x) = ub$ and $g(y) = ged(G,Q)$. Let z be the common ancestor of x and y, which is furthest from the root node. Let z' be a z's successor, which is also an ancestor of y. We have $f(z') \leq g(y) = ged(G,Q) < ub$.

Because z' is not in the path from z to x, it must be pruned during an iteration, that is, $f(z') \geq ub$ or $z'.visited = \text{true}$ (if z' has been inadmissibly pruned, we backtrack and re-expand it until it is either pruned or marked). The case when $f(z') \geq ub$ contradicts $f(z') < ub$, and for the other case $z'.visited = \text{true}$, we obtain $g(y) \geq ub$ using the same analysis as in Lemma 1, which contradicts $g(y) < ub$. Thus, $ub = ged(G,Q)$. □

Theorem 5. *Given two graphs G and Q and the beam width w, the space complexity of BSS_GED is $\mathcal{O}(w|V|^2)$, where $|V| = \max\{|V_G|, |V_Q|\}$.*

Proof. The storage space of BSS_GED consists of a beam stack bs, priority queues $open[0], \ldots, open[|V_G|]$, and a table new. Because the height of the search tree is $|V_G|$, the space of bs is $\mathcal{O}(|V_G|)$. For a layer l, $0 \leq l \leq |V_G|$, $open[l]$ stores at most w nodes; thus, the space of $open[l]$ is $\mathcal{O}(w)$. Therefore, all priority queues take $\mathcal{O}(w|V_G|)$ space.

A node r generates at most $(|V_Q| + 1)$ successors using GenSuccr; thus, $new[\mathcal{H}(r)]$ requires $\mathcal{O}(|V_Q|)$ space. Because all priority queues store at most $w(|V_G| + 1)$ nodes, new requires $\mathcal{O}(w \cdot (|V_G| + 1) \cdot |V_Q|)$ space. Combining the space of bs, $open[l]$ $(0 \leq l \leq |V_G|)$ and new, the space complexity of BSS_GED is $\mathcal{O}(|V_G| + w|V_G| + w|V_G||V_Q|) = \mathcal{O}(w|V|^2)$.

□

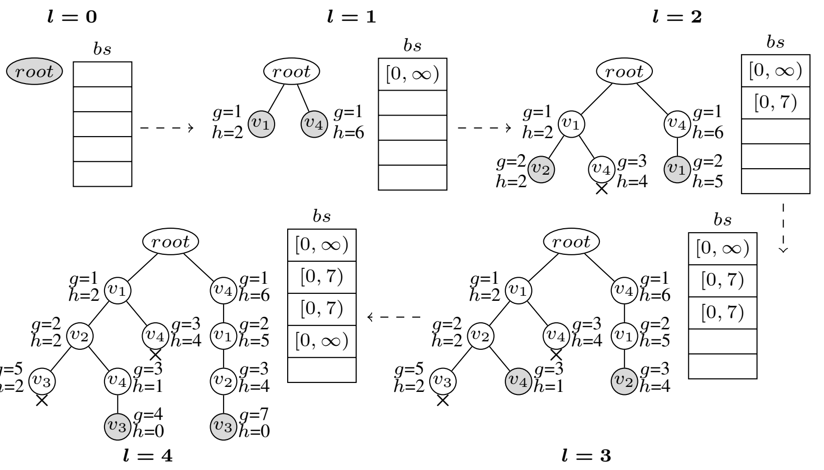

Figure 5: Example of computing GED with BSS_GED.

Example 6. Figure 5 shows how to use BSS_GED to compute $ged(G, Q)$, where G and Q are shown in Figure 1, and $w = 2$. "$\times$" implies that nodes are inadmissibly pruned, and the gray nodes denote the expanded nodes in each layer. The priority queues $open[0], \ldots, open[4]$ and table new are omitted due to space limitations. The values of g and h near a node give the incurred and estimated cost of this node, respectively.

In this example, starting from a dummy node $root$, we expand at most two nodes in each layer. In both layers 2 and 3, we generate three successors; thus, we inadmissibly prune the least desired node with the highest f-cost (i.e., the sum of g and h). When two nodes have the same f-cost, we preferentially prune the node with a higher g value. Correspondingly, we modify bs.top() from $[0, \infty)$ to $[0, 7)$. When we search to the last layer (i.e., $l = 4$), we obtain $ub = 4$. After that, we perform backtracking and pop items from bs. Because the right boundaries of all items in bs are greater than ub, bs will be empty when we stop backtracking. Finally, $ub = 4$ is the best edit cost we obtain–that is, $ged(G, Q) = 4$.

5. Search Space Pruning

In BSS_GED, when the f-cost of node r is greater than the upper bound ub, i.e., $f(r) = g(r) + h(r) \geq ub$, we can safely prune r. Because $g(r)$ is the irreversible cost, the upper bound ub and the lower bound $h(r)$ are the keys to perform pruning. In this section, we present two heuristics to prune the useless search space: (1) we propose an efficient heuristic function to estimate $h(r)$, and (2) we order the vertices in G to enable a fast search for a tight ub.

5.1. Estimating $h(r)$

A graph mapping ψ from G to Q induces an edit path $P_\psi : G \to G^\psi \to Q^\psi \to Q$ (see Section 2.2). Clearly, P_ψ contains at least $\max\{|V_G|, |V_Q|\} - |\Sigma_{V_G} \cap \Sigma_{V_Q}|$ vertex edit operations used to convert Σ_{V_G} to Σ_{V_Q}. Next we consider edit operations performed only on edges. Assume that we first delete γ_1 edges when converting G to G^ψ and then change γ_2 edge labels when converting G^ψ to Q^ψ, and finally, that we insert γ_3 edges when converting Q^ψ to Q.

First, when converting G to G^ψ by deleting γ_1 edges, we obtain $\Sigma_{E_{G^\psi}} \subseteq \Sigma_{E_G}$. Then, we change γ_2 edge labels when converting G^ψ to Q^ψ. Because in this transformation we need to change at least $|E_{Q^\psi}| - |\Sigma_{E_{G^\psi}} \cap \Sigma_{E_{Q^\psi}}|$ labels to convert $\Sigma_{E_{G^\psi}}$ to $\Sigma_{E_{Q^\psi}}$, we have $\gamma_2 \geq |E_{Q^\psi}| - |\Sigma_{E_{G^\psi}} \cap \Sigma_{E_{Q^\psi}}|$. Finally, we insert γ_3 edges when converting Q^ψ to Q; thus, $\Sigma_{E_{Q^\psi}} \subseteq \Sigma_{E_Q}$ and $\gamma_3 = |E_Q| - |E_{Q^\psi}|$. Clearly, we obtain $\gamma_2 + \gamma_3 \geq |E_Q| - |\Sigma_{E_{G^\psi}} \cap \Sigma_{E_{Q^\psi}}|$. Because $\Sigma_{E_{G^\psi}} \subseteq \Sigma_{E_G}$ and $\Sigma_{E_{Q^\psi}} \subseteq \Sigma_{E_Q}$, we have $|\Sigma_{E_G} \cap \Sigma_{E_Q}| \geq |\Sigma_{E_{G^\psi}} \cap \Sigma_{E_{Q^\psi}}|$. Therefore,

$$|\Sigma_{E_G} \cap \Sigma_{E_Q}| + \gamma_2 + \gamma_3 \geq |E_Q|. \tag{10}$$

We estimate the lower bounds of γ_1 and γ_3 below. For a vertex u in G, its degree d_u is the number of edges adjacent to u. The degree sequence $\delta_G = [\delta_G[1], \ldots, \delta_G[|V_G|]]$ of G is a permutation of $d_1, \ldots, d_{|V_G|}$ such that $\delta_G[i] \geq \delta_G[j]$ for $i < j$. When G and Q are of different sizes, we extend δ_G and δ_Q as $\delta'_G = [\delta_G[1], \ldots, \delta_G[|V_G|], 0_1, \ldots, 0_{|V|-|V_G|}]$ and $\delta'_Q = [\delta_Q[1], \ldots, \delta_Q[|V_Q|], 0_1, \ldots, 0_{|V|-|V_Q|}]$, respectively, where $|V| = \max\{|V_G|, |V_Q|\}$. Let $\Delta_1(G, Q) = \lceil \sum_{\delta'_G[i] > \delta'_Q[i]} (\delta'_G[i] - \delta'_Q[i])/2 \rceil$ and $\Delta_2(G, Q) = \lceil \sum_{\delta'_G[i] \leq \delta'_Q[i]} (\delta'_Q[i] - \delta'_G[i])/2 \rceil$, for $1 \leq i \leq |V|$. We have the following theorem.

Theorem 6 ([24]). *Given two graphs G and Q, $\gamma_1 \geq \Delta_1(G, Q)$ and $\gamma_3 \geq \Delta_2(G, Q)$.*

Based on Inequality (10) and Theorem 6, we know that $\sum_{i=1}^{3} \gamma_i \geq \Delta_1(G, Q) + |E_Q| - |\Sigma_{E_G} \cap \Sigma_{E_Q}|$ and $\sum_{i=1}^{3} \gamma_i \geq \Delta_1(G, Q) + \Delta_2(G, Q)$. Adding the edit operations performed on vertices, we then establish the lower bound of *ged*(G, Q) as follows.

Theorem 7. *Given two graphs G and Q, we have ged$(G, Q) \geq LB(G, Q)$, where $LB(G, Q) = \max\{|V_G|, |V_Q|\} - |\Sigma_{V_G} \cap \Sigma_{V_Q}| + \max\{\Delta_1(G, Q) + \Delta_2(G, Q), \Delta_1(G, Q) + |E_Q| - |\Sigma_{E_G} \cap \Sigma_{E_Q}|\}$.*

Consider a node $r = \{(u_{i_1} \to v_{j_1}), \ldots, (u_{i_l} \to v_{j_l})\}$ in the search tree. We divide G into two subgraphs G_1^r and G_2^r, where G_1^r is the mapped part of G such that $V_{G_1^r} = \{u_{i_1}, \ldots, u_{i_l}\}$ and $E_{G_1^r} = \{e(u, v) : u, v \in V_{G_1^r} \wedge e(u, v) \in E_G\}$, and G_2^r is the unmapped part such that $V_{G_2^r} = V_G \backslash V_{G_1^r}$ and $E_{G_2^r} = \{e(u, v) : u, v \in V_{G_2^r} \wedge e(u, v) \in E_G\}$. We obtain Q_1^r and Q_2^r similarly. Clearly, r induces an edit path P_r that transforms G_1^r to Q_1^r. Thus, the incurred cost from *root* to r, $g(r)$, is set to $|P_r|$ in BSS_GED.

For the unmapped parts G_2^r and Q_2^r, according to Theorem 7, we know that $LB(G_2^r, Q_2^r)$ is the lower bound of *ged*(G_2^r, Q_2^r); thus, we can adopt it as $h(r)$. However, $LB(G_2^r, Q_2^r)$ does not cover the potential edit cost on the edges between G_1^r (resp., Q_1^r) and G_2^r (resp., Q_2^r).

Definition 7 (Outer Edge Set). *The outer edge set of a vertex u in $V_{G_1^r}$ is defined as $O_u = \{e(u,v) : v \in V_{G_2^r} \wedge e(u,v) \in E_G\}$, which consists of edges adjacent to u that belong to neither $E_{G_1^r}$ nor $E_{G_2^r}$.*

Correspondingly, $O_{r(u)}$ is the outer edge set of $r(u)$, where $r(u)$ is the mapped vertex of u. Thus, $\Sigma_{O_u} = \{L(e(u,v)) : e(u,v) \in O_u\}$ is the label multiset of O_u. We assume that in the optimal edit path we first delete ξ_1^u edges and then change ξ_2^u edge labels, and finally, we insert ξ_3^u edges on u. Similar to the previous analysis of obtaining Inequality (10), we have

$$\begin{cases} |O_u| - \xi_1^u + \xi_3^u = |O_{r(u)}| \\ |\Sigma_{O_u} \cap \Sigma_{O_{r(u)}}| + \xi_2^u + \xi_3^u \geq |O_{r(u)}| \end{cases} \tag{11}$$

Clearly, we can find that $\sum_{i=1}^{3} \xi_i^u \geq \max\{|O_u|, |O_{r(u)}|\} - |\Sigma_{O_u} \cap \Sigma_{O_{r(u)}}|$ by Inequality (11). Adding all the vertices in G_1^r, we obtain the lower bound LB_1^r as follows:

$$\begin{aligned} LB_1^r = LB(G_2^r, Q_2^r) + \sum_{u \in V_{G_1^r}} (&\max\{|O_u|, |O_{r(u)}|\} \\ &-|\Sigma_{O_u} \cap \Sigma_{O_{r(u)}}|). \end{aligned} \tag{12}$$

Definition 8 (Outer Vertex Set). *The outer vertex set of a vertex u in G_1^r is defined as $A_u = \{v : v \in V_{G_2^r} \wedge e(u,v) \in E_G\}$, which consists of vertices in G_2^r adjacent to u.*

Correspondingly, $A_{r(u)}$ denotes the outer vertex set of $r(u)$. Thus, $A_G^r = \bigcup_{u \in V_{G_1^r}} A_u$ denotes the set of vertices in G_2^r adjacent to the outer edges between G_1^r and G_2^r. Similarly, we obtain $A_Q^r = \bigcup_{z \in V_{Q_1^r}} A_z$. When $|A_G^r| \leq |A_Q^r|$, we must insert at least $|A_Q^r| - |A_G^r|$ outer edges on some vertices in G_1^r; hence, $\sum_{u \in V_{G_1^r}} \xi_3^u \geq |A_Q^r| - |A_G^r|$ and $\sum_{u \in V_{G_1^r}} \xi_1^u \geq |A_G^r| - |A_Q^r|$ otherwise. Considering Inequality (11), for a vertex u in $V_{G_1^r}$, we have $\xi_2^u + \xi_3^u \geq |O_{r(u)}| - |\Sigma_{O_u} \cap \Sigma_{O_{r(u)}}|$. Thus, $\sum_{u \in V_{G_1^r}} (\xi_1^u + \xi_2^u + \xi_3^u) \geq \sum_{u \in V_{G_1^r}} (|O_{r(u)}| - |\Sigma_{O_u} \cap \Sigma_{O_{r(u)}}|) + \max\{0, |A_G^r| - |A_Q^r|\}$. Because $|O_u| + \xi_3^u = |O_{r(u)}| + \xi_1^u$, we have $\sum_{u \in V_{G_1^r}} (\xi_1^u + \xi_2^u + \xi_3^u) \geq \sum_{u \in V_{G_1^r}} (|O_u| - |\Sigma_{O_u} \cap \Sigma_{O_{r(u)}}|) + \max\{0, |A_Q^r| - |A_G^r|\}$. Therefore, we obtain the lower bounds, LB_2^r and LB_3^r, as follows:

$$\begin{aligned} LB_2^r = LB(G_2^r, Q_2^r) + \sum_{u \in V_{G_1^r}} &(|O_{r(u)}| - |\Sigma_{O_u} \cap \Sigma_{O_{r(u)}}|) \\ &+\max\{0, |A_G^r| - |A_Q^r|\}. \end{aligned} \tag{13}$$

$$\begin{aligned} LB_3^r = LB(G_2^r, Q_2^r) + \sum_{u \in V_{G_1^r}} &(|O_u| - |\Sigma_{O_u} \cap \Sigma_{O_{r(u)}}|) \\ &+\max\{0, |A_Q^r| - |A_G^r|\}. \end{aligned} \tag{14}$$

Based on LB_1^r, LB_2^r and LB_3^r, we adopt $h(r) = \max\{LB_1^r, LB_2^r, LB_3^r\}$ as the heuristic function in BSS_GED.

Example 7. Consider the search process of computing $ged(G, Q)$ in Figure 5, where G and Q are given in Figure 1. Given a node $r = \{(u_1 \to v_1), (u_2 \to v_2)\}$ (i.e., the leftmost node in the second layer), we obtain $G_2^r = (\{u_3, u_4\}, \{e(u_3, u_4)\}, L)$ and $Q_2^r = (\{v_3, v_4\}, \{e(v_3, v_4)\}, L)$. From Theorem 7, we can compute that $LB(G_2^r, Q_2^r) = 0$. For the processed vertices u_1 and u_2, we have $O_{u_1} = \{e(u_1, u_3)\}$, $O_{u_2} = \{e(u_2, u_4)\}$, $A_{u_1} = \{u_3\}$ and $A_{u_2} = \{u_4\}$. Clearly $\Sigma_{O_{u_1}} = \{\mathrm{b}\}$, $\Sigma_{O_{u_2}} = \{\mathrm{a}\}$ and $A_G^r = A_{u_1} \cup A_{u_2} = \{u_3, u_4\}$. Similarly we obtain $\Sigma_{O_{v_1}} = \{\mathrm{a}\}$, $\Sigma_{O_{v_2}} = \{\mathrm{a}\}$ and $A_Q^r = \{v_4\}$. Thus we can compute that $LB_1^r = LB(G_2^r, Q_2^r) + \sum_{u \in \{u_1, u_2\}} (\max\{|O_u|, |O_{r(u)}|\} - |\Sigma_{O_u} \cap \Sigma_{O_{r(u)}}|) = 1$. Because $A_G^r = \{u_3, u_4\}$ and $A_Q^r = \{v_4\}$, we obtain $LB_2^r = LB(G_2^r, Q_2^r) + \sum_{u \in \{u_1, u_2\}} (|O_{\psi(u)}| - |\Sigma_{O_u} \cap \Sigma_{O_{r(u)}}|) + \max\{0, |A_G^r| - |A_Q^r|\} = 2$, and $LB_3^r = LB(G_2^r, Q_2^r) + \sum_{u \in \{u_1, u_2\}} (|O_u| - |\Sigma_{O_u} \cap \Sigma_{O_{r(u)}}|) + \max\{0, |A_Q^r| - |A_G^r|\} = 1$. So, $h(r) = \max\{LB_1^r, LB_2^r, LB_3^r\} = \max\{1, 2, 1\} = 2$.

5.2. *Ordering Vertices in G*

In BSS_GED, we utilize GenSuccr to generate successors. Initially, we need to determine the processing order π of vertices in G (i.e., line 1 in Alg. 3). The most primitive way is to adopt the default vertex order $\pi = [u_1, \dots, u_{|V_G|}]$, which is also used in A*-GED [17] and DF-GED [26]. However, this order may be inefficient because it does not consider the structural relationships among vertices.

For two vertices u and v such that $e(u, v) \in E_G$, if u has been processed, then to obtain an early estimate of the edit cost on $e(u, v)$, we should process v as soon as possible. Hence our policy is to traverse G in a depth-first order to obtain π. However, starting the traversal from different vertices may result in different orders.

In Section 5.1, we proposed the heuristic function $h(r)$, an important part of which, $LB(G_2^r, Q_2^r)$, is presented in Theorem 7. As we know, the more structure that G_2^r and Q_2^r retain, the tighter $LB(G_2^r, Q_2^r)$ that we may obtain. Accordingly, we preferentially consider vertices with small degrees; this is because that when we first process those vertices, the remaining unmapped parts G_2^r and Q_2^r retain the structure to the greatest extent possible.

Algorithm 4 gives the method for computing the order π. First, we sort the vertices according to their degrees (line 2). Then, we call DFSTraverse to traverse G in a depth-first order (lines 3–6). When two vertices have the same degree, we preferentially consider the smaller vertex.

In the DFSTraverse procedure, we sequentially insert u into π and then mark u as visited (i.e., we set $F[u] = \mathrm{true}$) (line 1). Then, we compute N_u, the set of vertices adjacent to u (line 2). Finally, we select the unvisited vertex v from N_u with the smallest degree and then recursively call DFSTraverse to traverse the subtree rooted at v (lines 3–7).

Example 8. For the graph G shown in Figure 1, we first compute that $rank = [u_1, u_2, u_3, u_4]$. Starting from u_1, we traverse G in a depth-first order, and finally, we obtain $\pi = [u_1, u_2, u_4, u_3]$.

Algorithm 4: DetermineOrder(G)

```
1 $F[1..|V_G|] \leftarrow$ false, $\pi[] \leftarrow \emptyset$, $count \leftarrow 1$;
2 $rank \leftarrow$ sort vertices in $V_G$ according to their degrees;
3 for $i \leftarrow 1$ to $|V_G|$ do
4     $u \leftarrow rank[i]$;
5     if $F[u] =$ false then
6         DFSTraverse($u, F, rank, \pi, count$)
7 return $\pi$;
  procedure DFSTraverse($u, F, rank, \pi, count$)
1     $\pi[count] \leftarrow u, count \leftarrow count + 1, F[u] \leftarrow$ true;
2     $N_u \leftarrow \{v : v \in V_G \wedge e(u,v) \in E_G\}$;
3     while $|N_u| > 0$ do
4         $v \leftarrow \min\{rank[j] : j \in N_u\}$;
5         $N_u \leftarrow N_u \backslash \{v\}$;
6         if $F[v] =$ false then
7             DFSTraverse($v, F, rank, \pi, count$);
```

6. Experimental Results

In this section, we perform comprehensive experiments and then analyze the obtained results.

6.1. Datasets and Settings

We chose several real and synthetic datasets for use in the experiments. These datasets are described below.

- GREC [13]. The GREC dataset consists of 1,100 images, each of which represents a symbol from architecture, electronics, or another technical field. Vertices represent subparts of lines and are labeled with their types (circle, corner, endpoint, or intersection). Edges represent the connection points of lines and are labeled by line or arc.
- AIDS[2]. The AIDS dataset is an antiviral screening dataset from the NCI/NIH Development and Therapeutics Program and contains 42,687 chemical compounds. We generate labeled graphs from these compounds but omit the hydrogen atoms, as in [21, 24].
- Synthetic. The Synthetic dataset consists of dense graphs generated by the synthetic graph data generator GraphGen[3]. In the experiment, we generated the dataset S1K.E30.D30.L20, which contains 1000 graphs; the average number of

[2]http://dtp.nci.nih.gov/docs/aids/aidsdata.html
[3]http://www.cse.ust.hk/graphgen/

edges in each graph is 30; the density[4] of each graph is 30%; and the numbers of distinct vertex and edge labels are 20 and 5, respectively.

Tested datasets. Due to the difficulty of computing the GED, existing methods are unable to obtain the GED of large graphs within a reasonable amount of time. For the GREC and AIDS datasets, we excluded the large graphs (those with more than 35 vertices), as in [1], and then randomly selected 1,000 and 10,000 graphs to construct the datasets GREC-1K and AIDS-10K, respectively. For S1K.E30.D30.L20, we used the entire dataset.

Query Groups. In CSI_GED, for each tested dataset, given a specific graph size i, the authors suggested randomly choosing three data graphs whose sizes are $i-1$, i and $i+1$ to constitute the query group $i \pm 1$. We adopted the same approach in this study. Specifically, the query groups used in the experiment are $6 \pm 1, 9 \pm 1, 12 \pm 1, 15 \pm 1, 18 \pm 1$ and 21 ± 1.

Evaluation Metrics. Given a tested dataset $\mathcal{G} = \{\mathcal{G}_1, \mathcal{G}_2, \dots\}$ (e.g., AIDS-10K) and a query group $\mathcal{T} = \{\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3\}$ (e.g., 12 ± 1), for each tested algorithm (i.e., A*-GED, DF-GED, CSI_GED and BSS_GED), we perform $3 \times |\mathcal{G}|$ pairwise GED computations. Considering a GED computation of one graph pair $\mathcal{G}_i$ and $\mathcal{T}_j$, we set the available time and memory to 1 hour and 24 GB, respectively. When an algorithm required more than 1 hour or more than 24 GB to compute $ged(\mathcal{G}_i, \mathcal{T}_j)$, we set $slove(\mathcal{G}_i, \mathcal{Q}_j) = 0$; otherwise, we set $slove(\mathcal{G}_i, \mathcal{Q}_j) = 1$. Finally, we defined a metric named *solvable ratio* (sr) to evaluate the methods listed above. The sr is computed as follows:

$$sr = \frac{\sum_{j=1}^{3} \sum_{i=1}^{|\mathcal{G}|} slove(\mathcal{G}_i, \mathcal{T}_j)}{3 \times |\mathcal{G}|}. \tag{15}$$

Obviously, a larger sr reflects a better performance by an algorithm. In addition, we measured the average running time for processing these $3 \times |\mathcal{G}|$ pairwise GED computations. For a pairwise GED computation, when the running time of an algorithm exceeded 1 hour, we halted the algorithm and recorded the running time for this pairwise GED computation as 1 hour.

Environment. We conducted all the experiments on an HP Z800 PC running the Ubuntu 12.04 LTS operating system and equipped with a 2.67GHz GPU and 24 GB of memory. We implemented BSS_GED in C++, using -O3 to compile and run it. In BSS_GED, we set the beam width $w = 15$ for the sparse graphs in GREC-1K and AIDS-10K and $w = 50$ for the dense graphs in S1K.E30.D30.L20.

6.2. Comparing with Existing GED Computation Methods

In this section, we compare BSS_GED with state-of-the-art GED computation methods, including A*-GED [17], DF-GED [26] and CSI_GED [12]. Table 1 shows the solvable ratio (sr) and the average running time ($time$), where $\mathcal{G}$ and $\mathcal{T}$ denote the tested dataset and query group, respectively. "-OM" means that the memory usage

[4]the density of a graph G is defined as $\frac{2|E_G|}{|V_G|(|V_G|-1)}$.

Table 1: Solvable ratio and average running time of the tested methods.

$\mathcal{G}$	$\mathcal{T}$	A*-GED		DF-GED		CSI_GED		BSS_GED	
		sr	$time$	sr	$time$	sr	$time$	sr	$time$
GREC-1K	6 ± 1	100%	0.4 s	100%	81 ms	100%	**0.6 ms**	100%	0.8 ms
	9 ± 1	100%	11.9 s	100%	2.1 s	100%	5.6 ms	100%	**1.7 ms**
	12 ± 1	0%	-OM	56.7%	38.6 m	100%	96.3 ms	100%	**9.2 ms**
	15 ± 1	0%	-OM	3.1%	59.1 m	100%	1.3 s	100%	**0.2 s**
	18 ± 1	0%	-OM	0%	>1 h	100%	16.8 s	100%	**0.7 s**
	21 ± 1	0%	-OM	0%	>1 h	100%	46.5 s	100%	**2.1 s**
AIDS-10K	6 ± 1	100%	0.7 s	100%	94.2 ms	100%	**1.1 ms**	100%	3.1 ms
	9 ± 1	100%	27.3 s	100%	9.1 s	100%	**0.1 s**	100%	0.2 s
	12 ± 1	0%	-OM	42.2%	32.7 m	100%	2.5 s	100%	**1.9 s**
	15 ± 1	0%	-OM	0%	>1 h	100%	21.1 s	100%	**13.5 s**
	18 ± 1	0%	-OM	0%	>1 h	100%	1.2 m	100%	**0.5 m**
	21 ± 1	0%	-OM	0%	>1 h	100%	5.8 m	100%	**0.9 m**
S1K.E30.D30.L20	6 ± 1	100%	12.4 s	100%	2.4 s	100%	96 ms	100%	**17 ms**
	9 ± 1	100%	14.3 m	100%	3.2 m	100%	27.7 s	100%	**0.43 s**
	12 ± 1	0%	-OM	25.8%	48.1 m	69.1%	20.1 m	100%	**5.9 s**
	15 ± 1	0%	-OM	0%	>1 h	32.6%	52.1 m	100%	**1.1 m**
	18 ± 1	0%	-OM	0%	>1 h	0%	>1 h	100%	**12.3 m**
	21 ± 1	0%	-OM	0%	>1 h	0%	>1 h	75.4%	**37.5 m**

exceeds 24 GB. The abbreviations "ms", "s","m" and "h" represent milliseconds, seconds, minutes and hours, respectively.

From Table 1, we can see that BSS_GED completes almost all the pairwise GED computations and performs the best in terms of the sr. A*-GED cannot calculate the GED of graphs with more than 12 vertices within 24 GB, and DF-GED cannot finish the GED computation within 1 hour when the graphs have more than 15 vertices. CSI_GED performs well on the sparse graphs in GREC-1K and AIDS-10K; however, for the dense graphs in S1K.E30.D30.L20, the sr drops sharply as the query group size increases, confirming that it is unsuitable for dense graphs.

BSS_GED also achieves the best running time in most cases. DF-GED performs better than A*-GED, which is consistent with the previous results reported in [26]. Compared with DF-GED, BSS_GED is faster by at least two orders of magnitude. This improvement is because of several techniques used in BSS_GED, including the reduced search space, the efficient search paradigm, and the two heuristics. CSI_GED slightly outperforms BSS_GED on the sparse graphs in GREC-1K and AIDS-10K when the query group size is less than 9; in contrast, when the query group size is greater than 12, BSS_GED achieves 5x–20x and 1.5x–6x speedups on the above two datasets, respectively. Moreover, on the S1K.E30.D30.L20 dataset, BSS_GED achieves a 5x–220x speedup over CSI_GED. These results show that BSS_GED is highly efficient on both sparse and dense graphs.

6.3. *Evaluating BSS_GED*

As described earlier in this paper, we adopt several techniques in BSS_GED, including (1) GenSuccr, which is used to decrease the number of invalid and redundant mappings (Section 3); (2) the beam-stack search, which is used to establish a trade-off between memory utilization and backtracking calls (Section 4); and (3) two heuristics, where one estimates the node's cost (Section 5.1) and the other is a vertex sorting strategy (Section 5.2). Thus, it is necessary to study the contributions of these techniques to BSS_GED.

In the following tests, we fix the query group 12 ± 1 as the tested query group. Then, we separately evaluate the techniques listed above. Note that even when we are evaluating one technique, the other techniques are still included in BSS_GED.

6.3.1. Evaluating GenSuccr

In this section, we evaluate the effect of GenSuccr on the performance of BSS_GED. To implement the comparison, we replace GenSuccr with three other methods: BasicGenSuccr, GenSuccr-R1 and GenSuccr-R2, to generate a node's successors. BasicGenSuccr is the basic method used in A*-GED [17] and DF-GED [26]. GenSuccr-R1 (resp., GenSuccr-R2) denotes that we only use Rule 1 (resp., Rule 2). Table 2 reports the obtained results, where $\#nodes$ shows the average number of nodes generated during the GED computation and $time$ shows the average running time. Here, we do not display the sr for the four methods because they are all 100%.

Table 2: Average number of nodes generated and average running time.

Methods	GREC-1K		AIDS-10K		S1K.E30.D30.L20	
	$\#nodes$	$time$	$\#nodes$	$time$	$\#nodes$	$time$
BasicGenSuccr	4.23×10^3	11.64 ms	5.99×10^5	2.28 s	1.38×10^6	8.63 s
GenSuccr-R1	3.96×10^3	10.69 ms	5.53×10^5	2.13 s	1.01×10^6	6.03 s
GenSuccr-R2	3.54×10^3	10.25 ms	5.24×10^5	2.07 s	1.37×10^6	8.58 s
GenSuccr	3.26×10^3	9.21 ms	5.02×10^5	1.94 s	0.99×10^6	5.89 s

From Table 2, we can clearly see that GenSuccr generates the smallest number of nodes and that both GenSuccr-R1 and GenSuccr-R2 perform better than BasicGenSuccr. On the GREC-1K and AIDS-10K datasets, GenSuccr-R2 performs better than GenSuccr-R1, which means that more redundant mappings than invalid mappings can be reduced. In contrast, for S1K.E30.D30.L20, GenSuccr-R1 performs better. Compared to BasicGenSuccr, GenSuccr generates 23%, 16% and 27% fewer nodes on GREC-1K, AIDS-10K and S1K.E30.D30.L20, respectively; accordingly, GenSuccr results in speedups of 25%, 18% and 32% on the above three datasets, respectively, compared to BasicGenSuccr. Thus, using GenSuccr we create a reduced search space.

6.3.2. Evaluating Beam-stack Search

In this section, we evaluate the effect of the beam-stack search (BSS) by comparing it with two other search paradigms: the best-first search used in A*-GED and the depth-first search (DFS) used in DF-GED. Note that DFS is a special case of BSS when $w = 1$. In addition, we also select four different w values (w = 5, 15, 50 and 100) to evaluate the effect of w on the performance of BSS.

Figure 6 reports the obtained results. The left subfigure shows the average memory usage of different search paradigms, and the right subfigure gives the average number of backtracking calls. For A*, we do not show the number of backtracking calls because it does not incur backtracking. Note that when we measure the average

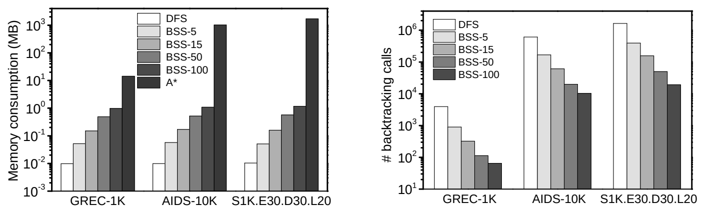

Figure 6: Average memory usage and number of backtracking calls.

memory usage of A* for a pairwise GED computation, if A* takes more than 24 GB, then the computation fails, and we record this computation's memory usage as 24 GB.

From Figure 6, clearly, A* requires considerably more memory than DFS and BSS. DFS stores only the nodes of a path from the root to a leaf node; thus, it consumes the least memory but incurs the greatest number of backtracking calls. Although BSS consumes more memory than DFS, its memory usage is typically small (e.g., < 3 MB) and can be ignored; however, it substantially reduces the number of backtracking calls. For example, when $w = 15$, the number of backtracking calls in BSS is only 10% of that in DFS.

Table 3: Solvable ratio and average running time.

Search	GREC-1K		AIDS-10K		S1K.E30.D30.L20	
	sr	$time$	sr	$time$	sr	$time$
A*	100%	93.1 ms	98.2%	8.34 s	95.9%	18.79 s
DFS	100%	12.5 ms	100%	3.47 s	100%	11.41 s
BSS-5	100%	9.8 ms	100%	2.15 s	100%	8.43 s
BSS-15	100%	**9.2 ms**	100%	**1.94 s**	100%	7.29 s
BSS-50	100%	11.2 ms	100%	2.53 s	100%	**5.89 s**
BSS-100	100%	14.9 ms	100%	2.93 s	100%	7.94 s

In addition, we show the sr and the average running time of the different search paradigms in Table 3. Clearly, A* is unable to complete all the pairwise computations and suffers from the highest running time. This result occurs because the priority queue used in A* stores numerous nodes, and updating the priority queue each time is also computationally expensive. In contrast, the update costs of DFS and BSS are much lower. Moreover, both DFS and BSS can quickly find a leaf node and use the obtained upper bound to prune nodes in the subsequent search. Compared to DFS, BSS runs faster because it requires fewer backtracking calls. In conclusion, BSS consumes more memory than DFS but incurs fewer backtracking calls; this means that BSS achieves a trade-off between memory utilization and backtracking calls.

One additional trend to consider is that as w increases, the running time first decreases and then increases; the running time is maximized when $w = 15$ on the GREC-1K and AIDS-10K datasets and when $w = 50$ on the S1K.E30.D30.L20 dataset. Two factors may contribute to this trend: (1) When w is too small, BSS becomes trapped in a local suboptimal solution, resulting in many backtracking calls, and (2) when w is too large, BSS may unnecessarily expand too many nodes in each layer.

6.3.3. *Evaluating $h(r)$*

In BSS_GED, we proposed a new heuristic function $h(r)$ to prune the useless search space. In this section, we evaluate $h(r)$ by comparing it with two other heuristics: LS_1 and LS_2. LS_1 used in both A*-GED and DF_GED is the label multiset lower bound[5]. LS_2 used in CSI_GED also computes the degree distance, but it ignores the edge labels. Figure 7 shows the solvable ratio and the average running time.

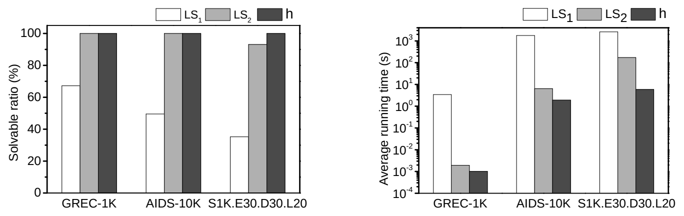

Figure 7: Solvable ratio and average running time.

Clearly, our heuristic function estimates a tighter lower bound and provides more pruning ability than LS_1 and LS_2. LS_1, which only considers the labels and ignores the structure information, performs the worst. In contrast, LS_2 is more similar to our approach, as it also uses the degree information; however, it ignores the edge labels. Compared with LS_2, our heuristic achieves speedups of 1.7x, 3x and 50x on the GREC-1K, AIDS-10K and S1K.E30.D30.L20 datasets, respectively. Thus, we can conclude that the proposed heuristic function efficiently prunes the useless search space.

6.3.4. *Evaluating DetermineOrder*

The other heuristic that we proposed is the vertex sorting strategy discussed in Section 5.2. In this section, we evaluate this heuristic by comparing it with a basic method (i.e., no sorting). Figure 8 shows the average number of generated nodes and the average running time. We do not show the sr because both versions achieve 100%.

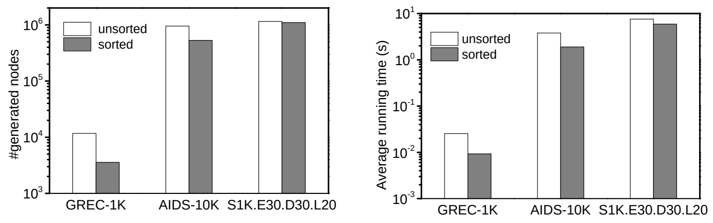

Figure 8: Average number of nodes generated and running time.

From Figure 8, we can see that the sorting method generates fewer nodes and requires less time. Specifically, compared to the unsorted method, sorting decreases the number of generated nodes by 65%, 40% and 7% on the GREC-1K, AIDS-10K and S1K.E30.D50.L20 datasets, respectively. Accordingly, the sorting method achieves speedups of about 2x, 1.3x and 0.1x on the above three datasets, respectively. Thus, we can conclude that using the sorting strategy further improves BSS_GED's efficiency.

[5]Both A*-GED and DF_GED use the bipartite graph matching to compute the minimum label substitution cost. In fact, its unweighted version is the label multiset lower bound.

7. Extension of BSS_GED

In this section, we investigate extending BSS_GED to solve the graph similarity search problem: Given a graph database $\mathcal{G} = \{\mathcal{G}_1, \mathcal{G}_2, \dots\}$, a query graph $\mathcal{Q}$, and a threshold τ, the problem aims to find all graphs in $\mathcal{G}$ satisfying $ged(\mathcal{G}_i, \mathcal{Q}) \leq \tau$. Almost all the well-known solutions to this problem [22, 24, 27, 28, 29, 30] are based on the filtering-and-verification schema–that is, they first filter $\mathcal{G}$ to obtain candidate graphs and then verify the candidate graphs using expensive GED computations.

We also use the same strategy. For each data graph $\mathcal{G}_i$, we first compute the lower bound $LB(\mathcal{G}_i, \mathcal{Q})$ from Theorem 7. If $LB(\mathcal{G}_i, \mathcal{Q}) > \tau$, then $ged(\mathcal{G}_i, \mathcal{Q}) \geq LB(\mathcal{G}_i, \mathcal{Q}) > \tau$; hence, we filter $\mathcal{G}_i$. Otherwise, $\mathcal{G}_i$ becomes a candidate graph.

To verify each candidate graph $\mathcal{G}_i$, we need to compute $ged(\mathcal{G}_i, \mathcal{Q})$. The basic approach is to first compute $ged(\mathcal{G}_i, \mathcal{Q})$ and then determine whether $\mathcal{G}_i$ is a required graph by judging $ged(\mathcal{G}_i, \mathcal{Q}) \leq \tau$. By incorporating τ within BSS_GED, we can further accelerate the verification. First, we set the initial upper bound ub to $\tau + 1$ (line 3 in Alg. 3). Then, during the execution of BSS_GED, when we reach a leaf node r, if r's cost (i.e., $g(r)$) satisfies $g(r) \leq \tau$, then $\mathcal{G}_i$ must be a required graph, and we terminate BSS_GED. The underlying reason for this is that $g(r)$ is an upper bound of $ged(\mathcal{G}_i, \mathcal{Q})$; hence, we have $ged(\mathcal{G}_i, \mathcal{Q}) \leq g(r) \leq \tau$. When verifying all the candidate graphs, we obtain the final required graphs.

7.1. Performance Evaluation on Graph Similarity Search

In this section, we evaluate the performance of BSS_GED as a standard graph similarity search query method. To perform the comparison, we select two other methods GSimJoin [28] and CSI_GED [12][6]. GSimJoin is a path-based *q-gram* indexing method and filters data graphs based on a *q-gram* counting lower bound and a label multiset lower bound; it uses indexing to accelerate the filtering process and adopts a modified A*-GED as its verifier. Similar to BSS_GED, CSI_GED also provides support for graph similarity search.

In this experiment, we used the full datasets GERC, AIDS and S1K.E30.D30.L20 and randomly selected 100 graphs from each dataset as the query graphs. We varied the threshold τ from 2 to 12 to evaluate the performance of GSimJoin, CSI_GED and BSS_GED. Table 4 reports the total running time by each method under different τ values, where GJ, CG and BG are short for GSimJoin, CSI_GED and BSS_GED, respectively.

From Table 4, clearly, BSS_GED achieves the best performance in most cases, especially when τ is large. GSimJoin is unable to finish the graph similarity search when $\tau \geq 8$ on the GREC and AIDS datasets because its verifier A*-GED depletes the 24 GB of memory. Compared with GSimJoin for τ values where it can finish, BSS_GED achieves speedups of 280x–75000x, 6.5x–15000x and 4x–3000x on the GREC, AIDS and S1K.E30.D30.L20 datasets, respectively. CSI_GED performs

[6]In CSI_GED, the author ignored other indexing-based graph similarity search query methods such as κ-AT [22] and Mixed [29] because the experimental results had confirmed that CSI_GED performed far better than those methods. Here we also used the same setting.

Table 4: Total running time on the complete test datasets.

τ	GREC			AIDS			S1K.E30.D30.L20		
	GJ	CG	BG	GJ	CG	BG	GJ	CG	BG
2	4.7 m	1.0 s	**1.0 s**	36.8 s	**3.4 s**	5.6 s	0.4 s	0.2 s	**0.1 s**
4	4.3 h	17.3 s	**2.6 s**	43.6 m	15.2 s	**10.8 s**	0.9 s	0.4 s	**0.2 s**
6	227.5 h	6.7 m	**11.1 s**	265.9 h	2.4 m	**1.1 m**	1.8 s	0.6 s	**0.4 s**
8	-OM	1.5 h	**1.4 m**	-OM	1.1 h	**8.5 m**	64.1 s	3.2 s	**0.6 s**
10	-OM	6.4 h	**11.5 m**	-OM	96.7 h	**1.9 h**	31.4 m	42.9 s	**2.1 s**
12	-OM	32.3 h	**57.6 m**	-OM	262.4 h	**25.9 h**	8.5 h	32.2 m	**10.3 s**

slightly better than BSS_GED when $\tau = 2$ on the AIDS dataset. In contrast, BSS_GED performs much better when $\tau \geq 4$ and the gap between them widens as τ increases; specifically, BSS_GED achieves speedups of 30x–60x, 2x–50x and 1.1x–180x compared to CSI_GED on the GREC, AIDS and S1K.E30.D30.L20 datasets, respectively. Thus, we can conclude that BSS_GED efficiently finishes the graph similarity search and runs much faster than the existing methods.

8. Related Works

Recently, the GED computation has received considerable attention. Riesen et al. [17] proposed the first standard method, A*-GED, based on the best-first search paradigm. A*-GED organizes all possible graph mappings as an ordered search tree, where the inner nodes denote partial mappings and the leaf nodes denote complete mappings. Provided that the heuristic function estimates the lower bound of the GED, A*-GED guarantees that the first complete mapping found yields the GED. However, A*-GED needs to store numerous partial mappings, resulting in a huge memory consumption; in practice, it can only address small graphs. To overcome this bottleneck, Abu-Aisheh et al. [26] proposed a depth-first search method, DF-GED, whose memory requirement increases linearly with the number of vertices of the compared graphs. In addition, DF-GED employs a branch-and-bound strategy to prune the useless search space.

In contrast to A*-GED and DF-GED, Gouda et al. [12] proposed a novel edge-based mapping method, CSI_GED, for computing the GED. The core idea of CSI_GED is to enumerate all possible common isomorphic substructures between the compared graphs, where each common substructure corresponds to an edit path. CSI_GED organizes all the enumerated substructures as an edge-based mapping search tree and applies the depth-first search paradigm to traverse this search tree. In addition, CSI_GED proposed three heuristics to accelerate the search. Empirical results showed that CSI_GED achieved an excellent performance on sparse graphs. Because CSI_GED only works for the uniform model, Blumenthal et al. [1] generalized it to cover the non-uniform model.

In addition, Justice et al. [14] first formulated the GED computation as a binary linear programming (BLP) problem, therein searching for the permutation matrix that minimizes the cost of transforming one graph to another; however, their solution has two limitations: it only considers unweighted graphs, and it is unable to process graphs with edge labels. Lerouge et al. [31] developed a new BLP formulation to overcome Justice's limitations and solved the BLP by calling a commercial solver.

Another work closely related to GED computation is the graph similarity search problem, namely, finding data graphs in the database that are similar to a given query graph within a threshold. Due to the difficulty of computing the GED, most graph similarity search methods [22, 24, 27, 28, 29, 30] adopt a filtering-and-verification strategy to speed up the graph similarity search. In the filtering phase, the GED lower bounds are employed to prune as many false positive graphs from the database as possible. The remaining unpruned graphs constitute a candidate set and are validated with expensive GED computations in the verification phase. Using the same strategy, BSS_GED can easily be extended to solve the graph similarity search problem. Note that most graph similarity search methods adopt $\mathsf{A}^{\star}$-GED as their verifier; because BSS_GED outperforms $\mathsf{A}^{\star}$-GED for the GED computation, BSS_GED can also be considered as a verifier to accelerate them.

9. Conclusions and Future Work

In this paper, we present a novel vertex-based mapping method BSS_GED, for computing the GED of two labeled graphs. BSS_GED can be efficiently applied to sparse and dense graphs, benefiting from two aspects: (1) It decreases many invalid and redundant mappings during the GED computation and hence creates a reduced search space; (2) It employs a novel search paradigm, *beam-stack search*, to establish a trade off between memory utilization and expensive backtracking calls. Extensive experiments showed that BSS_GED performed better than the state-of-the-art methods.

As a measure, the GED has been widely served as a foundation in many applications such as graph similarity search, image recognition, and protein structure classification. Therefore, efficient GED computation is critical. For instance, in the area of graph similarity search, the verification phase of computing the GED is the most time-consuming; thereby, BSS_GED can be employed to accelerate this phase, and experiments of applying BSS_GED to the graph similarity search described in Section 7 have shown its efficiency. Alternatively, in the context of kernel machines (e.g., SVM and PCA), many methods consider the GED as a similarity measure to classify various objects represented by graphs; BSS_GED can also be used to improve those methods' classification efficiency.

However, computing the GED is an NP-hard problem [27]; thus it is quite difficult to obtain the GED of two large graphs within a reasonable amount of time. One possible direction is to employ parallel processing techniques to accelerate the GED computation. On the other hand, many works [16, 23, 32] focus on the suboptimal solution for calculating the GED. How to apply BSS_GED to quickly obtain a high-quality suboptimal solution of GED is left for future work.

10. Acknowledgments

The authors thank Kaspar Riesen and Zeina Abu-Aisheh for providing their source files and Karam Gouda and Xiang Zhao for providing their executable files. This work was supported in part by the China NSF grants 61741215, 61373044 and 61173025, and by the Fundamental Research Funds for the Central Universities of

China grant JBF180301, and by the US NSF grant CCF-1017623. Hongwei Huo is the corresponding author.

References

[1] D. B. Blumenthal and J. Gamper. Exact computation of graph edit distance for uniform and non-uniform metric edit costs. In *GbRPR*, pages 211–221, 2017.

[2] J. E. Beasley and N. Christofides. Theory and methodology: Vehicle routing with a sparse feasibility graph. *Eur. J. Oper. Res.*, 98(3):499–511, 1997.

[3] R. Ibragimov, M. Malek, J. Baumbach, and J. Guo. Multiple graph edit distance: simultaneous topological alignment of multiple protein-protein interaction networks with an evolutionary algorithm. In *GECCO*, pages 277–284, 2014.

[4] D. J. Watts, P. S. Dodds, and M. E. J. Newman. Identity and search in social networks. *Science*, 296(5571):1302–1305, 2002.

[5] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *ICDM*, pages 721–724, 2002.

[6] D. Conte, P. Foggia, C. Sansone, and M. Vento. Thirty years of graph matching in pattern recognition. *Int. J. Pattern Recogn.*, 18(3):265–298, 2004.

[7] M. Neuhaus and H. Bunke. Edit distance based kernel functions for structural pattern classification. *Pattern Recogn.* 39(10):1852–1863, 2006.

[8] P. Mahé, N. Ueda, T. Akutsu, J. L. Perret, and J. P. Vert. Graph kernels for molecular structureactivity relationship analysis with support vector machines. *J. Chem. Inf. Model.*, 45(4):939–951, 2005.

[9] E. E. Schadt, S. H. Friend, and D. A. Shaywitz. A network view of disease and compound screening. *Nat. Rev. Drug Discov.*, 8(4):286–295, 2009.

[10] N. Acosta-Mendoza, A. Gago-Alonso and J. E. Medina-Pagola. Frequent approximate subgraphs as features for graph-based image classification. *Knowl.-Based Syst.*, 27(3):381–392, 2012.

[11] J. Xuan, J. Lu, G. Zhang, and X. Luo. Topic Model for Graph Mining. *IEEE Trans. Cybernetics*, 45(12):2792–2803, 2015.

[12] K. Gouda and M. Hassaan. CSI_GED: An efficient approach for graph edit similarity computation. In *ICDE*, pages 256–275, 2016.

[13] K. Riesen and H. Bunke. IAM graph database repository for graph based pattern recognition and machine learning. In *SSPR*, pages 287–297, 2008.

[14] D. Justice and A. Hero. A binary linear programming formulation of the graph edit distance. *IEEE Trans. Pattern Anal Mach Intell.*, 28(8):1200–1214, 2006.

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

[16] K. Riesen and H. Bunke. Approximate graph edit distance computation by means of bipartite graph matching. *Image Vision Comput.*, 27(7):950–959, 2009.

[17] K. Riesen, S. Emmenegger, and H. Bunke. A novel software toolkit for graph edit distance computation. In $GbRPR$, pages 142–151, 2013.

[18] R. M. Marín, N. F. Aguirre, and E. E. Daza. Graph theoretical similarity approach to compare molecular electrostatic potentials. *J. Chem. Inf. Model.*, 48(1):109–118, 2008.

[19] S. Russell and P. Norvig. Artificial intelligence: a modern approach (2nd ed.). Prentice-Hall, 2002.

[20] R. Zhou and E. A. Hansen. Beam-stack search: Integrating backtracking with beam search. In $ICAPS$, pages 90–98, 2005.

[21] D. W. Williams, J. Huan, and W. Wang. Graph database indexing using structured graph decomposition. In $ICDE$, pages 976–985, 2007.

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

[23] A. Fischer, K. Riesen, and H. Bunke. Improved quadratic time approximation of graph edit distance by combining Hausdorff matching and greedy assignment. *Pattern Recogn Lett.*, 87:55–62, 2017.

[24] X. Chen, H. Huo, J. Huan, and J. S. Vitter. MSQ-Index: A succinct index for fast graph similarity search. *arXiv preprint arXiv:1612.09155*, 2016.

[25] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In $SIGMOD$, pages 335–346, 2004.

[26] Z. Abu-Aisheh, R. Raveaux, J. Y. Ramel, and P. Martineau. An exact graph edit distance algorithm for solving pattern recognition problems. In $ICPRAM$, pages 271–278, 2015.

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

[28] X. Zhao, C. Xiao, X. Lin, and W. Wang. Efficient graph similarity joins with edit distance constraints. In $ICDE$, pages 834–845, 2012.

[29] W. Zheng, L. Zou, X. Lian, D. Wang, and D. Zhao. Efficient graph similarity search over large graph databases. *IEEE Trans. Knowl Data Eng.*, 27(4):964–978, 2015.

[30] Y. Liang and P. Zhao. Similarity Search in Graph Databases: A Multi-Layered Indexing Approach. In $ICDE$, pages 783–794, 2017.

[31] J. Lerouge, Z. Abu-Aisheh, R. Raveaux, P. Héroux, and S. Adam. New binary linear programming formulation to compute the graph edit distance. *Pattern Recogn.*, 72:254–265, 2017.

[32] K. Gouda, M. Arafa, and T. Calders. A novel hierarchical-based framework for upper bound computation of graph edit distance. *Pattern Recogn.*, 80:210–224, 2018.