BFST_ED: A novel upper bound computation framework for the graph edit distance

Karam Gouda [†‡], Mona Arafa [†], and Toon Calders [‡]

†Faculty of Computers & Informatics, Benha University, Egypt.
‡Computer & Decision Engineering department, Universit Libre de Bruxelles, Belgium
{karam.gouda,mona.arafa}@fci.bu.edu.eg
{karam.gouda,toon.calders}@ulb.ac.be

Abstract. Graph similarity is an important operation with many applications. In this paper we are interested in graph edit similarity computation. Due to the hardness of the problem, it is too hard to exactly compare large graphs, and fast approximation approaches with high quality become very interesting. In this paper we introduce a novel upper bound computation framework for the graph edit distance. The basic idea of this approach is to picture the comparing graphs into hierarchical structures. This view facilitates easy comparison and graph mapping construction. Specifically, a hierarchical view based on a breadth first search tree with its backward edges is used. A novel tree traversing and matching method is developed to build a graph mapping. The idea of spare trees is introduced to minimize the number of insertions and/or deletions incurred by the method and a lookahead strategy is used to enhance the vertex matching process. An interesting feature of the method is that it combines vertex map construction with edit counting in an easy and straightforward manner. This framework also allows to compare graphs from different hierarchical views to improve the upper bound. Experiments show that tighter upper bounds are always delivered by this new framework at a very good response time.

Keywords: Graph similarity, graph edit distance, upper bounds.

1 Introduction

Due to its ability to capture attributes of entities as well as their relationships, graph data model is currently used to represent data in many application areas. These areas include but are not limited to Pattern Recognition, Social Network, Software Engineering, Bio-informatics, Semantic Web, and Chem-informatics. Yet, the expressive power and flexibility of graph data representation model come at the cost of high computational complexity of many basic graph data tasks. One of such tasks which has recently drawn lots of interest in the research community is computing the graph edit distance. Given two graphs, their graph edit distance computes the minimum cost graph editing to be performed on one of them to get the other. A graph edit operation is a kind of vertex insertion/deletion, edge insertion/deletion or a change of vertex/edge's label (relabeling) in the graph.

A close relationship exists between graph editing and graph mapping. Given a graph editing one can define a graph mapping and vice versa. The problem of graph edit distance computation is then reduced to the problem of finding a graph mapping which induces a minimum edit cost. Graph edit distance computation methods such as those

based on A* [6, 13, 12] exploit this relationship and compute graph edit distance by exploring the vertex mapping space in a best first fashion in order to find the optimal graph mapping. Unfortunately, since computing graph edit distance is NP-hard problem [16] those methods can not scale to large graphs. In practice, to be able to compare large graphs, fast algorithms seeking suboptimal solutions have been proposed. Some of them deliver unbounded solutions [14, 15, 1, 17], while others compute either upper and/or lower bound solutions [9, 16, 2, 4].

Recent interesting upper bounds and the one introduced in this paper are obtained based on graph mapping. The intuition is that the better the mapping between graphs, the better the upper bound on their edit distance. In [10] a graph mapping method is developed, which first constructs a cost matrix between the vertices of the two graphs, and then uses a cubic-time bipartite assignment algorithm, called Hungarian algorithm [8], to optimally match the vertices. The cost matrix holds the matching costs between the neighbourhoods of corresponding vertices. The idea behind this heuristic being that a mapping between vertices with similar neighborhoods should induce a graph mapping with low edit cost. A similar idea is used in [16]. The main problem with these heuristics is that the pairwise vertex cost considers the graph structure only locally. Thus, in cases where neighborhoods do not differentiate the vertices, e.g., as with unlabeled graphs, these methods work poorly. To enhance the graph mapping obtained by these methods and tighten the upper bound, additional search strategies were deployed, however, at the cost of extra computation time. For example, an exhaustive vertex swapping procedure is used in [16]. A greedy vertex swapping is used in [11]. Even though much time is needed by these improvements, the resulted graph mapping is prone to local optima, which is susceptible to initialization.

This paper presents a novel linear-time upper bound computation framework for the graph edit distance. The idea behind this approach is to picture the comparing graphs into hierarchical structures. This view facilitates easy comparison and graph mapping construction. To implement the framework, the breadth first search tree (BFST) representation is adopted as a hierarchical view of the graph, where each comparing graph is represented by a breadth first search tree with its backward edges. A pre-order BFST traversing and matching method is then developed in order to build a graph mapping. A slight drift from the pure pre-order traversal is that for each visited source vertex in the traversal, all its children and those of its matching vertex are matched before visiting any of these children. This facilitates for a vertex to find a suitable correspondence to match among various options. In addition, the idea of spare trees is introduced to decrease the number of insertions and/or deletions incurred by the method, and a lookahead strategy is used to enhance the vertex matching process. An interesting feature of the matching method is that it combines map construction with edit counting in easy and straightforward manner. This novel framework allows to explore a quadratic space of graph mappings to tighten the bound, where for each two corresponding vertices it is possible to run the tree traversing and matching method on the distinct hierarchical view imposed by these two vertices. Moreover, this quadratic space can be explored in parallel to speed up the process, a feature which is not offered by any of the previous methods. Experiments show that tighter upper bounds are always delivered by this framework at a very good response time.

2 Preliminaries

2.1 Graphs

In this section, we first give the basic notations. Let Σ be a set of discrete-valued labels. A labeled graph G can be represented as a triple (V, E, l), where V is a set of vertices, $E \subseteq V \times V$ is a set of edges, and l: $V \rightarrow \Sigma$ is a labeling function. $|V|$ is the numbers of vertices in G, and is called the *order* of G. The degree of a vertex v, denoted $deg(v)$, is the number of vertices that are directly connected to v. A labeled graph G is said to be *connected*, if each pair of vertices $v_i, v_j \in V$, $i \neq j$, are directly or indirectly connected. In this paper, we focus on simple and connected graphs with labeled vertices. A simple graph is undirected graph with neither self-loops nor multiple edges. Hereafter, a labeled graph is simply called a graph unless stated otherwise.

A graph $G = (V, E, l)$ is a *subgraph* of another graph $G' = (V', E', l')$, 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 $G \cong G'$.*

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

2.2 Graph Editing and Graph Edit Distance

Given a graph G, a graph edit operation p is a kind of vertex or edge deletion, a vertex or edge insertion, or a vertex relabeling. Notice that vertex deletion occurs only for isolated vertices. Each edit operation p is associated with a cost $c(p)$ to do it depending on the application at hand. It is clear that a graph edit operation transforms a graph into another one. A sequence of edit operations $\langle p_i \rangle_{i=1}^{k}$ performed on a graph G to get another graph G' is called *graph editing*, denoted $G^{edit} = \langle p_i \rangle_{i=1}^{k}$. The cost of graph editing is, thus, the sum of its edit operation's costs, i.e. $\mathcal{C}(G^{edit}) = \sum_{i=1}^{k} c(p_i)$.

Given two graphs G_1 and G_2 there could be multiple graph editings of G_1 to get G_2. The optimal graph editing is defined as the one associated with the minimal cost among all other graph editings transforming G_1 into G_2. The cost of an optimal graph editing defines the *edit distance* between G_1 and G_2, denoted $GED(G_1, G_2)$. That is, $GED(G_1, G_2) = \min_{G^{edit}} \mathcal{C}(G^{edit})$. In this paper we assume the unit cost model, i.e. $c(p) = 1, \forall p$. Thus, the optimal graph editing is the one with the minimum number of edit operations.

Example 1 *Fig. 1 shows two graphs G_1 and G_2. An optimal graph editing of G_1 to get G_2 can be obtained as follows: A deletion operation of the edge (u_1, u_2), a relabeling operation of the vertex u_3 from label B into label C, an insertion of a new vertex u_5 with label C, and an insertion of a new edge (u_5, u_4). Thus, $GED(G_1, G_2) = 4$.*

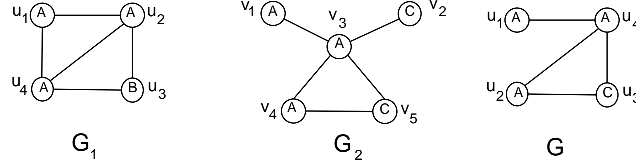

Fig. 1. Two comparing graphs G_1 and G_2.

2.3 Graph Mapping

Given two graphs G_1 and G_2, a *graph mapping* aims at finding correspondence between the vertices and edges of the two graphs. Every vertex map $f\colon V_1 \cup \{u^n\} \rightarrow V_2 \cup \{v^n\}$, where u^n and v^n are dummy vertices with special label ϵ, defines a graph mapping, where the vertex $u \in V_1$ or $v \in V_2$ has no correspondence at the other graph if $f(u) = v^n$ or $f(u^n) = v$, resp. The edge $(u, v) \in E_1$ has no correspondence if $(f(u), f(v)) \notin E_2$. Also, the edge $(v, v') \in E_2$ has no correspondence if $(u, u') \notin E_1$ such that $v = f(u)$ and $v' = f(u')$.

There exists a relationship between graph editing and graph mapping. More generally any graph mapping induces a graph editing which relabels all mapped vertices, and inserts or delete the non-mapped vertices/edges of the two graphs [5]. Conversely, given a graph editing, the maximum common subgraph isomorphism (MCSI) between G and G_2 defines a graph mapping between G_1 and G_2, where G is the graph obtained from G_1 after applying the deletion and relabeling operations in the graph editing.

Example 2 *Given the graph editing of Example 1. The graph G obtained from G_1 after applying the edge deletion and vertex relabeling operations of this graph editing is shown in Fig. 1. The MCSI $f = \{(u_1, v_1),\ (u_2, v_4),\ (u_3, v_5),\ (u_4, v_3)\}$ between G and G_2 defines a graph mapping. On the other hand, consider the vertex map $f = \{(u_1, v_2), (u_2, v_4), (u_3, v_1), (u_4, v_3), (u^n, v_5)\}$. A graph editing can be defined from f as: two relabeling operations on u_1 and u_3. Two deletion operations of the edges (u_1, u_2) and (u_2, u_3). An insertion operation of a vertex corresponding to the unmatched vertex v_5. Two insertion operations of the edges (u_4, u_5) and (u_2, u_5).*

In view of the relationship between graph editing and graph mapping, the problem of graph edit distance computation is reduced to the problem of finding an optimal graph mapping – a mapping which induces a minimum edit cost. Due to the hardness of obtaining such a graph mapping (computing graph edit distance is known to be NP-hard problem [16]), approximate graph mapping methods become very popular, especially when large graphs are under investigation [16, 11, 7, 3]. Any of those mapping methods overestimates the graph edit distance. The intuition behind those methods is that the better the mapping between the comparing graphs, the better the upper bound on their edit distance. In this paper we present an efficient upper bound computation framework for the graph edit distance which is also based on graph mapping. We first sketch the framework and then present the details of the implementing algorithm. 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).

3 A Novel Upper Bound Computation Framework

The main idea of our approach is to picture the graphs to be compared into hierarchical structures. This view allows easy comparison and fast graph mapping construction. It also facilitates counting of the induced edit operations. Breadth first search (BFS) is a graph traversing method allowing a hierarchical view of the graph through the breadth first search tree it constructs. This view is defined as follows.

Definition 3. (BFST representation of a graph) *Given a graph G and a vertex $u \in G$. Let T_u be the breadth first search tree (BFST) rooted at u. The BFST representation of G given u, denoted as G^u, is defined by the BFST-Edges pair $G^u = \langle T_u, E_u \rangle$, where E_u is the set of graph edges which are not part of T_u, called backward edges.*

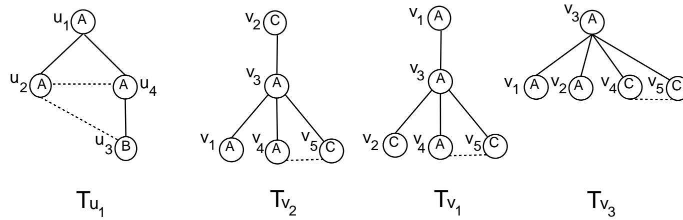

Fig. 2. One BFST view for G_1, $G_1^{u_1} = \langle T_{u_1}, E_{u_1} \rangle$, and three different for G_2, namely, $G_2^{v_2} = \langle T_{v_2}, E_{v_2} \rangle$, $G_2^{v_1} = \langle T_{v_1}, E_{v_1} \rangle$ and $G_2^{v_3} = \langle T_{v_3}, E_{v_3} \rangle$. Black edges constitute BFSTs and backward edges are shown by dashed lines.

Example 3 *Consider the graphs G_1 and G_2 of Fig. 1. Fig. 2 shows some of their hierarchical representations using breadth first search trees.*

Algorithm 1: `BFST_ED`(G_1, G_2)

1: Let T_u and T_v be the breadth first trees rooted at $u \in G_1$ and $v \in G_2$, resp;
2: $f = \{0, \ldots, 0\}$; $f_{cost} = 0$; /* f is a vertex map*/
3: `BFST_Mapping_AND_Cost`(T_u, T_v, f, f_{cost});
4: **for** each source or target backward edge **do**
5: **if** the matching vertices of its end points have no backward edge **then** f_{cost}++;
6: output f and f_{cost};

Fig. 3. `BFST_ED`: An upper bound computation framework of $GED(G_1, G_2)$.

Given the source and target graphs G_1 and G_2. Let T_u and T_v be the breadth first trees rooted at $u \in G_1$ and $v \in G_2$, resp. Based on the BFST view of the graph, an upper bound computation framework of the graph edit distance can be developed. First a tree mapping between T_u and T_v is constructed. This tree mapping determines a vertex map between the vertex sets of the two graphs. Using this vertex map, the edit cost on backward edges is calculated and then added to the tree mapping edit cost to produce an upper bound of the graph edit distance. Note that it is possible as a result of the tree matching method an edge is inserted at the position of a source backward edge. If it is the case the final edit cost should be decremented because an edge is already there and this insertion should not be occurred. This framework, named BFST_ED (which stands for the bold letters in: Breadth First Search Tree based Edit Distance), is outlined in

Fig. 3. The vector f holds the map on graph vertices. The value $f_i \neq 0$ indicates that the ith vertex of V_1 has been mapped. f_{cost} is the graph mapping cost.

The most important step in this framework is the tree mapping and edit counting method `BFST_Mapping_AND_Cost`. The better the tree mapping produced by this routine, the better the overall graph edit cost returned by the framework. The question now is *how to build a good tree mapping between two breadth first search trees?* In the following subsections we answer this question.

3.1 Random & Degree-based BFSTs Matching

The simplest and most direct answer to the previous question is to randomly match vertices at corresponding tree levels. That is, a source vertex at a given tree level l can match any target vertex at the corresponding level. This matching, however, may incur a huge edit cost between the two trees as a vertex having no correspondence has to be deleted as well as its subtree if it is a source one, or to be inserted with its subtree if it is a target one.[1] Moreover, any of these subtree insertions or deletions entails the insertion or deletion of an edge connecting the subtree with its parent. Unfortunately, the number of vertices that have no correspondence will increase as we go down the tree using this matching method. Suppose that at a given tree level the number of source vertices is equal to the number of target ones, and at one of its preceding levels, there exist vertices with no correspondence. Deletions or insertions of subtrees made at the preceding tree level will change the equality at the given level and entail extra deletions and/or insertions.

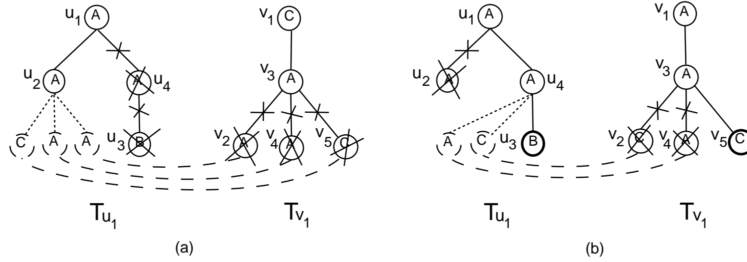

Fig. 4. A picture of the edit operations performed on two comparing BFSTs (a) using random assignment (b) using OUT degree assignment. Vertex/edge insertions are shown by dashed vertices/edges. Vertex relabeling is done on blacked source vertices.

Example 4 *Given the source and target trees T_{u_1} and T_{v_1} of Fig. 2. The edit cost returned by BFST_ED is 13. The random matching in* `BFST_Mapping_AND_Cost` *induces 10 edit operations, and 3 edit operations are required for backward edge modifications. The vertex map returned by* `BFST_Mapping_AND_Cost` *is as:* $f = \{(u_1, v_1), (u_2, v_3), (u_3, v^n), (u_4, v^n), (u^n, v_2), (u^n, v_4), (u^n, v_5)\}$. *This map includes 2 vertex deletions, 2 edge deletions, 3 vertex insertions, and 3 edge insertions. Fig. 4 (a) gives a picture on how the* `mapping_AND_cost` *method based on random assignment matches the source T_{u_1} with the target T_{v_1} and computes their graph edit cost.*

[1] Since all edit modifications usually occur at the source tree to get the target one, any deletion at the target tree is equivalent to an insertion at the source tree in our model.

An idea to decrease the number of insertions and/or deletions caused by random assignment, and thus decrease the overestimation of GED, is based on the **OUT** degree of a BFST vertex defined as follows.

Definition 4. (OUT degree of a BFST vertex) *Given a graph G. Let T_u be the BFST rooted at $u \in G$. For each tree vertex $w \in T_u$, the OUT degree of w, denoted $OUT(w)$, is defined as the number of its children in the tree.*

The idea is to match the vertices at corresponding tree levels which have near OUT degrees. According to this matching, vertices which have no correspondence will decrease and consequently the edit cost returned by the method as well. Based on this idea, the edit cost in Example 4 is decreased from 13 to 10 edit operations as the vertex map returned by `BFST_Mapping_AND_Cost` has four less insertion and deletion operations, two on vertices and two on edges, at the cost of one extra vertex relabeling operation for matching the source vertex at the bottom level. The associated vertex map is given as follows: $f = \{(u_1, v_1), (u_4, v_3), (u_2, v^n), (u_3, v_2), (u^n, v_4), (u^n, v_5)\}$. This map incurs 7 edit operations on the BFSTs and 3 on backward edges. Fig. 4 (b) pictures the tree editing based on OUT degree assignment.

Although this matching method is very fast,[2] still the overall edit cost returned is far from the graph edit distance. In the running example, the best edit cost returned is 10 which is large compared with 4 – the graph edit distance. Another important issue of this matching method which is not seen by the running example is that the method is not taking care of the matching occurred for parents while matching children. It may happen that for many matched children, their parents are matched differently which requires extra edit operations. Though this counting can be accomplished in a subsequent phase using the associated vertex map, the tree mapping cost will be very high. Next, we present a tree mapping and matching method addressing all previous issues.

3.2 An Efficient BFSTs Matching Method

The bad overestimation of the graph edit distance returned by the previous method is due to two reasons. One lies at the simple tree traversing method which does not take previous vertex matching into account and blindly processes the trees level by level. The second reason lies at the vertex matching process itself: Vertices are randomly matched or in the best case are matched based on their OUT degrees which offer a very narrow lookahead view for the comparing vertices. Not to mention the very large number of insertions and/or deletions produced by this matching method. Below we introduce a new tree traversal and vertex matching method which addresses all previous issues.

Traversing the comparing BFSTs in pre-order can offer a solution to the first issue as vertices can be matched in the traversal order. This matching order guarantees that vertices can be matched only if their parents are matched. Though the pre-order traversal removes the overhead of any subsequent counting phase as in the previous method, it limits the different options for matching a given vertex, where only one option is allowed which is based on the visited vertex. To overcome this, one can compare and

[2] No computations are soever required for random assignment; only climbing the source tree and at each tree level the corresponding vertices are randomly matched. For OUT degree assignment, extra computations are required to match vertices with the closest OUT degrees.

match all corresponding children of both an already visited source vertex and its matching target before visiting any of these children. This in turn facilitates for a child to find a suitable correspondence to match among various options.

What is the suitable correspondence for a vertex to match? It could be based on the OUT degree as in the previous method. However, the OUT degree gives a very narrow view as we have already noticed. Fortunately, the BFST structure offers a wider lookahead view which is adopted by our method. This view is represented by a tuple, called feature vector, consisting of three values attached with each vertex. These values are calculated during the building process of the BFSTs.

Definition 5. (A feature vector of a BFST vertex) *Given a graph G and let T_u be the BFST rooted at $u \in G$. For each tree vertex $w \in T_u$, the feature vector of w, denoted $f(w)$, is a tuple $f(w) = \langle SUB(w), BW(w), l(w)\rangle$, in which:*

- *$SUB(w)$ is the number of vertices and edges of the subtree rooted at w.*
- *$BW(w)$ is the number of backward edges incident on w.*
- *$l(w)$ is the vertex label.*

Obviously, all tree leaves have SUB count zero. $BW(w)$ is defined for each tree vertex w as: $BW(w) = deg(w) - (OUT(w) + 1)$. Based on Definition 5, a source vertex favors a target vertex to match which has near vertex distance, defined as follows.

Definition 6. (Vertex distance) *Given two source and target tree vertices w and w' with their feature vectors $f(w)$ and $f(w')$. The distance between w and w', denoted $d(w, w')$, is defined based on feature vectors as:*

$$d(w, w') = |SUB(w) - SUB(w')| + |BW(w) - BW(w')| + c(w, w'), \quad (1)$$

where the cost function c returns 0 if the two matching items, i.e. vertices w and w', have identical labels, and 1 otherwise.

By considering the difference $|BW(w) - BW(w')|$ in calculating the vertex distance, the method partially takes care of the backward edges while matching vertices. In fact, $BW(w)$ is introduced to minimize the number of edit operations required for matching backward edges. Formally, let $C_u = \{u_1, \ldots, u_k\}$ and $C_v = \{v_1, \ldots, v_l\}$ be the children of two matched source and target vertices u and v, in the given order. A child u_i of u favors a child v_k of v to match based on the following equation.

$$k = argmin_{v_j \in C_v}(d(u_i, v_j)). \quad (2)$$

That is, the distance between a vertex u_i and its matching vertex v_k should be minimal among other vertices. In cases where there are more than one candidate for a vertex to match, then the method selects the one with the smallest vertex id.

So far the preorder traversal with Equation 2 addresses some of the previous issues: No subsequent counting phase is required by the method and the method also offers a wider lookahead view to better match the corresponding vertices. Unfortunately, this traversal may worsen the other issues. In fact it may increase the number of insertions and/or deletions because it could happen that for a visited vertex the number of its children differs from the number of children of its matching vertex, though the total number of vertices might be equal at the children level. To overcome this issue the idea of *spare trees* is brought to the method.

Definition 7. **(spare subtrees)** *Given two comparing BFSTs T_u and T_v rooted at $u \in G_1$ and $v \in G_2$, resp. Any subtree of T_u or T_v rooted at a vertex w is called spare subtree if the vertex w has no correspondence while pre-order traversing T_u and T_v.*

The idea of spare subtrees has been introduced in order to answer the following question: *Why do we get rid of each unmatched vertex with its subtree and pay a high edit cost for doing so, though it could be beneficial later on instead of being costly right now.* The pre-order traversing and matching method is developed by building a *spare-parts store ST_u* at each comparing BFST T_u in order to preserve these unmatched vertices and their subtrees. During tree traversal, when an encountered source or target vertex has no correspondence, the method asks the spare-parts store for a suitable counterpart. If such a spare-part does exist it is matched and removed from the store, otherwise the new vertex itself with its subtree goes to the relevant spare-parts store. This idea guarantees that each vertex will get a counterpart as long as the other tree has this counterpart, i.e., if the number of vertices of the other tree has at least the number of vertices of the tree where the vertex belongs to. At the end of the tree traversal the spare-parts store associated with the tree of small order will be empty and the other store will contain a number of spare subtrees equal to the vertex difference $||V_1| - |V_2||$. Finally, the number of vertices and edges in each remaining spare subtree will be added to the tree mapping cost. Fortunately, the size of each remaining spare subtree will be very small.

Algorithm 2 in Fig. 5 is a recursive encoding of the method. In fact we do not put the whole spare subtrees in the store, references to their roots are the only information that is maintained (refer to line 12). Also, if a vertex and its subtree is characterized as a spare part, the connecting edge with its parent vertex (the vertex where it hangs on) is deleted and the tree mapping cost is updated (see line 3: All edges connecting children which have no correspondence are deleted if they are source vertices and inserted otherwise). Moreover, if this vertex is a source one, it is temporary blocked, i.e., it is temporary removed from the pre-order traversal (line 13). Alternatively, if a spare source subtree is matched and removed from the store, it goes directly into the pre-order traversal again (line 28). It means that the root of this subtree will be hung on and become a child of the currently processing parent vertex. For hanging this spare vertex no edge insertion is required since the matching vertex, whether it comes from the other spare store or as a corresponding child, has already charged by an equivalent deletion operation at line 3 of the edge connecting it with its parent.

Example 5 *Fig. 6 explains how the traversing method (Algorithm 2) matches T_{u_1} with T_{v_1} of Fig. 2 and computes the tree edit cost. The graph edit cost produced by BFST_ED is 5: 6 edit operations are required to transform T_{u_1} into T_{v_1}; one of the tree edge insertions is removed because it is occurred at the position of the backward edge (u_2, u_4), and finally zero edit operations are required on the remaining backward edges.*

Theorem 1. *(Time Complexity) The procedure* `BFST_mapping_AND_cost` *(Algorithm 2) returns the vertex map f and its induced edit cost f_{cost} in $O(d^2|V_1|)$, where d is the maximum vertex degree in both graphs.*

Theorem 2. *(Correctness) The value f_{cost} returned by* `BFST_ED`*(G_1, G_2) with Algorithm 2 at Fig. 5 is the edit cost induced by the returned vertex map f.*

Algorithm 2: `mapping_AND_cost`$(T_{r_1}, T_{r_2}, f, f_{cost})$

1: let $C_{r_1} = \{u_{11}, \ldots, u_{1n_1}\}$ be the children of r_1 in the given order;
2: let $C_{r_2} = \{u_{21}, \ldots, u_{2n_2}\}$ be the children of r_2 in the given order;
3: $f_{cost} += ||C_{r_1}| - |C_{r_2}||$; /* edge deletion if $|C_{r_1}| > |C_{r_2}|$, insertions otherwise */
4: Let C_{r_l} be the smallest set of children, $l = 1$ or 2 (in the following steps $m = 1$ or 2, $m \neq l$)
5:   **for** each child $u_{li} \in C_{r_l}$ **do** /*find a suitable correspondence from C_{r_m}.*/
6: k = argmin$_{u_{mj} \in C_{r_m}} d(u_{li}, u_{mj})$;
7:   **if** $l = 2$ **then** $f[u_{mk}] = u_{li}$;
8:   **else** $f[u_{li}] = u_{mk}$; u_{mk} is matched;
9:   **if** $l(u_{li}) \neq l(u_{mk})$ **then** f_{cost}++;
10:   **for** each remaining $u_{mi} \in C_{r_m}$ **do** /* each u_{mi} searches for correspondence at ST$_{r_l}$.*/
11:   **if** ST$_{r_l} = \emptyset$ **then** /*u_{mi} is spared if the other store is empty */
12:   ST$_{r_m}$ = ST$_{r_m} \cup \{u_{mi}\}$;
13: $C_{r_m} = C_{r_m} \setminus \{u_{mi}\}$;
14:   **else** /*u_{mi} tries to find a suitable correspondence.*/
15: k = argmin$_{u_j \in ST_{r_l}} d(u_{mi}, u_j)$;
16:   **if** $l = 2$ **then** $f[u_{mi}] = u_k$;
17:   **else**
18: $f[u_k] = u_{mi}$; u_{mi} is matched;
19: C_{r_l} = $C_{r_l} \cup \{u_k\}$; /*if u_k is a source one, it goes into the preorder traversal again.*/
20:   **if** $l(u_k) \neq l(u_{mi})$ **then** f_{cost}++;
21:   ST$_{r_l}$ = ST$_{r_l} \setminus \{u_k\}$;
22: **for** each $u_{1i} \in C_{r_1}$ **do**
23:   `mapping_AND_cost`$(T_{u_{1i}}, T_{f[u_{1i}]}, f, f_{cost})$;

Fig. 5. Pre-order traversing and matching method.

3.3 Improving the Overestimation: BFST_ED_ALL

Previously, based on the chosen graph vertex, a hierarchical representation of the graph could be given. Thus, for each graph G, it is possible to construct $|V|$ distinct hierarchical views, each of which starts from a different vertex. The multi-hierarchical views of a graph gives us the opportunity to compare two graphs from different hierarchical perspectives and choose the best obtained graph mapping, instead of restricting ourselves to a single view comparison. This multi-view comparison is implemented and called BFST_ED_ALL. In fact BFST_ED_ALL explores $|V_1| \times |V_2|$ possible graph mappings and returns the one with the least overestimation.

4 Experimental Evaluation

In this section, we aim at empirically studying the proposed method. We conducted several experiments, and all experiments were performed on a 2.27GHz Core i3 PC with 4GB memory running Linux. Our method is implemented in standard C++ using the STL library and compiled with GNU GCC.

Benchmark Datasets: We chose several real graph datasets for testing the method.
1) **AIDS** (http://dtp.nci.nih.gov/docs/aids/aidsdata.html) is a DTP AIDS Antiviral Screen chemical compound dataset. It consists of 42, 687 chemical compounds, with an average of 46 vertices and 48 edges. Compounds are labelled with 63 distinct vertex labels but the majority of these labels are H, C, O and N.

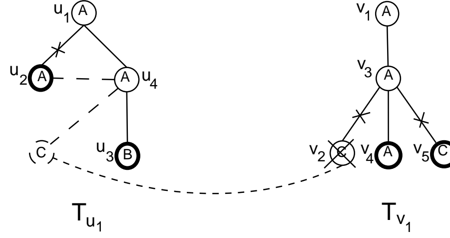

Fig. 6. A possible tree editing transforming T_{u_1} into T_{v_1}, which is produced by the preorder method (algorithm 2). Vertex and edge insertions are shown by dashed lines and vertex relabeling is shown by heavy-blacked lines. This tree editing has the following 6 edit operations given in order according to the algorithm: deletion of the edge (u_1, u_2), deletion of two target edges which is equivalent to two edge insertions at the source tree, relabeling of u_3, deletion of v_2 which is equivalent to vertex insertion at the source tree, and relabeling of u_2. The vertex map returned by this algorithm in the order of its construction is as follows: $f = \{(u_1, v_1), (u_4, v_3), (u_3, v_4), (u_2, v_5), (u^n, v_2)\}$.

2) **Linux** (`www.comp.nus.edu.sg/~xiaoli10/data/segos/linux_segos.zip`) is a Program Dependence Graph (PDG) dataset generated from the Linux kernel procedure. PDG is a static representation of the data flow and control dependency within a procedure. In the PDG graph, an vertex is assigned to one statement and each edge represents the dependency between two statements. PDG is widely used in software engineering for clone detection, optimization, debugging, etc. The Linux dataset has in total 47,239 graphs, with an average of 45 vertices each. The graphs are labelled with 36 distinct vertex labels, representing the roles of statements in the procedure, such as declaration, expression, control-point, etc.
3) **Chemical** is a chemical compound dataset. It is a subset of PubChem (`pubchem.ncbi.nlm.nih.gov`) and consists of one million graphs. It has 24 vertices and 26 edges on average. The graphs are labelled with 81 distinct vertex labels.

4.1 Comparison With Exact Methods

We first evaluate the performance of our methods, BFST_ED and BFST_ED_All, against exact GED computation methods. We want to see how much speed up can be achieved by our methods at the cost of how much loss in accuracy of GED. In this experiment, we use the recent exact GED computation method named CSI_GED [5], and randomly choose two source and target vertices to run BFST_ED. As the exact computation of GED is expensive on large graphs, to make this experiment possible, graphs with acceptable order were randomly selected from the data sets. From these graphs, four groups of ten graphs each were constructed. The graphs in each group have the same number of vertices, and the number of vertices residing in each graph among different groups varies from 5 to 20. In this experiment, each group is compared with the one having the largest graph order. Thus, we have 100 graph matching operations in each group comparison. For estimating the errors, the mean relative overestimation of the exact graph edit distance, denoted ϕ_o, is calculated.[3] Fig. 7 plots the value ϕ_o of each

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

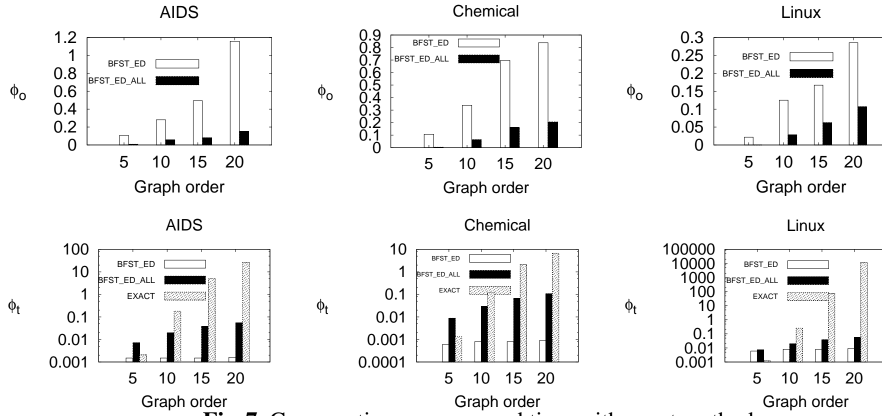

Fig. 7. Comparative accuracy and time with exact method.

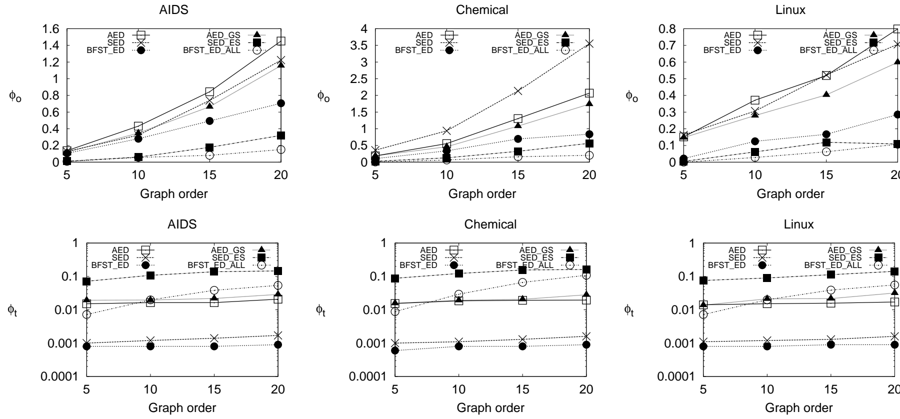

Fig. 8. Comparative accuracy and time with different methods: Small order graphs.

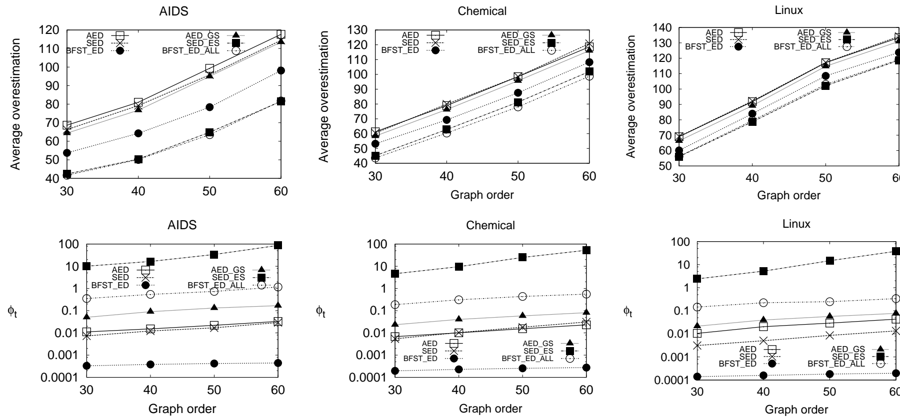

Fig. 9. Comparative accuracy and time with different methods: Large order graphs

method on each group for the different data sets, where the horizontal axis shows the order of the comparing group. It is clear that ϕ_o = 0 for CSI_GED. Fig. 7 also plots the mean run time ϕ_t taken by each method on each group for each data set.

First, we observe that on the different data sets the accuracy loss of BFST_ED_All is very small on small order groups and increases with increasing graph order. It is between 10-20% on large groups. Accuracy loss of BFST_ED, on the other hand, is even worse and exhibits the same trend. It is about 3-4 times larger than that of BFST_ED_All. Looking at the run time of the three methods. We observe that on large groups comparisons, BFST_ED_All outperforms CSI_GED by 2-5 orders of magnitude and it is outperformed by BFST_ED from 1-2 orders of magnitude. One thing that should be noticed is that on the very small order group, the one with order 5, CSI_GED is faster than BFST_ED_All on all real dat sets.

4.2 Comparison With Approximation Methods

In this set of experiments, we compare our methods against the state-of-the-art upper bound computation methods such as Assignment Edit Distance (AED) method [10], the Star-based Edit Distance (SED) method [16], and their extensions. These methods are extended by applying a postprocessing vertex swapping phase to enhance the obtained graph mapping. In [5], a greedy vertex swapping procedure is applied on the map obtained from AED, and is abbreviated as "AED_GS", and in [16] an exhaustive vertex swapping is applied on the map obtained from SED and is abbreviated as "SED_ES". The executables for competitor methods were obtained from their authors.

Comparison With Respect to GED First we compare the different methods on graphs where the exact graph edit distance is known. Therefore, we use the groups of graphs from the previous experiment. To look at bound tightness, ϕ_o is calculated for each of these methods. Obviously, the smaller the mean relative overestimation, the better is the approximation method. We also aim at investigating ϕ_t for each method.

Fig. 8 plots ϕ_o and ϕ_t for each method on the different data sets. It shows that `BFST_ED_All` always produces smaller ϕ_o values than the ones produced by other methods on all data sets. The gap between ϕ_o values is remarkable on the AIDS and Chemical data sets, where ϕ_o values of `BFST_ED_All` are almost half of those produced by SED_ES, the best competitor. On Linux data set, those produced by SED_ES are comparable with ours on the largest group comparison. In addition to the good results on bound tightness, the average run time of `BFST_ED_ALL` is better than that of other methods. It is about 2 times faster than the best competitor. Looking at each method individually, there is a clear trade-off between bound tightness and speed. The first map is always come at high speed but at the cost of accuracy loss. In conclusion, we can see that the upper bound obtained by `BFST_GED_ALL` provides near approximate solutions at a very good response time compared with current methods.

Comparison on Large Graphs In this set of experiments we evaluate the different methods on large graphs. In each data set, four groups of ten graphs each are selected randomly, where each group has a fixed graph order chosen as: 30, 40, 50, and 60. Each of these groups is compared using the different methods with a database of 1000 graphs chosen randomly from the same data set. Fig. 9 shows the average edit overestimation returned by each method per graph matching on each group. The average edit overestimation is adopted instead of ϕ_o since there is no reference GED value available for large graphs. The figure also shows the average running time for all data sets.

Fig. 9 shows that both AED and SED have the same accuracy on all data sets with almost the same running time (except that AED is two times faster on Linux). AED_GS

shows little improvements of accuracy over AED with time increase. BFS_ED, on the other hand, shows much better accuracy with 2-3 orders of magnitude speed up over the previous three methods. Also, both BFST_ED_All and SED_ES show the same accuracy on all data sets; but with two orders of magnitude speed up for the benefit of BFST_ED_All. These results shows the scalability of our methods on large graphs.

5 Conclusion

In this paper, the computational methods approximating the graph edit distance are studied; in particular, those overestimating it. A novel overestimation approach is introduced. It uses breadth first hierarchical views of the comparing graphs to build different graph maps. This approach offers new features not present in the previous approaches, such as the easy combination of vertex map construction and edit counting, and the possibility of constructing graph maps in parallel. Experiments show that near overestimation is always delivered by this new approach at a very good response time.

References

1. Conte, D., Foggia, P., Sansone, C., Vento, M.: Thirty years of graph matching in pattern recognition. Int. J. Pattern Recognit. Artif. Intell. **18**, 265–298 (2004)
2. Fischer, A., Suen, C., Frinken, V., Riesen, K., Bunke, H.: Approximation of graph edit distance based on hausdorff matching. Pattern Recognition **48(2)**, 331–343 (2015)
3. Gauzere, B., Bougleux, S., Riesen, K., Brun, L.: Approximate graph edit distance guided by bipartite matching of bags of walks. In: S+SSPR, pp. 73–82 (2014)
4. Gouda, K., Arafa, M.: An improved global lower bound for graph edit similarity search. Pattern Recognition Letters **58**, 8–14 (2015)
5. Gouda, K., Hassaan, M.: CSI_GED: An efficient approach for graph edit similarity computation. In: ICDE, pp. 265–276 (2016)
6. Hart, P., Nilsson, N., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. SSC **4(2)**, 100–107 (1968)
7. Justice, D., Hero, A.: A binary linear programming formulation of the graph edit distance. IEEE Trans. PAMI **28(8)**, 1200–1214 (2006)
8. Munkres, J.: A network view of disease and compound screening. J.Soc. Ind. Appl.Math. **5**, 32–38 (1957)
9. Neuhaus, M., Bunke, H.: Edit distance-based kernel functions for structural pattern classification. Pattern Recognition **39**, 1852–1863 (2006)
10. Riesen, K., Bunke, H.: Approximate graph edit distance computation by means of bipartite graph matching. Image Vis. Comput. **27(7)**, 950–959 (2009)
11. Riesen, K., Bunke, H.: Improving approximate graph edit distance by means of a greedy swap strategy. In: ANNPR, pp. 129–140 (2014)
12. Riesen, K., Emmenegger, S., Bunke, H.: A novel software toolkit for graph edit distance computation. In: GbRPR, pp. 142–151 (2013)
13. Riesen, K., Fankhauser, S., Bunke, H.: Speeding up graph edit distance computation with a bipartite heuristic. In: MLG, pp. 21–24 (2007)
14. Riesen, K., Neuhaus, M., Bunke, H.: Bipartite graph matching for computing the edit distance of graphs. In: GbRPR, pp. 1–12 (2007)
15. Serratosa, F.: Fast computation of bipartite graph matching. Pattern Recognition Letters **45**, 244–250 (2014)
16. Zeng, Z., Tung, A., Wang, J., Feng, J., Zhou, L.: Comparing stars: On approximating graph edit distance. PVLDB **2(1)**, 25–36 (2009)
17. Zhao, X., Xiao, C., Lin, X., Wang, W., Ishikawa, Y.: Efficient processing of graph similarity queries with edit distance constraints. VLDB J. **22**, 727–752 (2013)