Received September 10, 2020, accepted September 22, 2020, date of publication September 25, 2020, date of current version October 6, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3026648

HGED: A Hybrid Search Algorithm for Efficient Parallel Graph Edit Distance Computation

JONGIK KIM
Division of Computer Science and Engineering, Jeonbuk National University, Jeonju 54896, South Korea
e-mail: jongik@jbnu.ac.kr

This work was supported in part by the Research Funds of Jeonbuk National University, in 2020, and in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Korea Government (Ministry of Science and ICT) under Grant 2019R1F1A1059795.

ABSTRACT Graph edit distance (GED) is a measure for quantifying the similarity between two graphs. Because of its flexibility and versatility, GED is widely used in many real applications. However, the main disadvantage of GED is its high computational cost. Many solutions have been proposed to speed up GED computation, but most of them focus on developing serial algorithms and only very few solutions consider parallel computing. In this paper, we study a parallel GED computation to elaborate a fast and precise algorithm. Unlike existing solutions that utilize either a depth-first or a best-first search, we propose a hybrid approach that combines depth-first and best-first search paradigms. Our approach can quickly find tighter GED upper bounds, and effectively prune the search space using the upper bounds. Based on the approach, we develop an efficient parallel GED computation algorithm named HGED. To maximize thread utility, HGED is also equipped with a novel dynamic load balancing scheme whose main focus is on reducing the overhead of thread synchronization. Experimental results on widely used real datasets show that, on average, HGED outperforms the state-of-the art serial algorithm AStar^{+}-LSa by 6 times and the state-of-the parallel algorithm PGED by 4 times.

INDEX TERMS Graph similarity, graph edit distance, parallel computation, hybrid search, dynamic load balancing.

I. INTRODUCTION

Graphs have been used in a wide spectrum of applications to represent entities and relationship/interaction between entities. The availability a graph similarity measure is a fundamental requirement in those applications. To quantify the similarity between graphs, therefore, various similarity measures have been developed, such as maximum common subgraphs [1], [2], missing edges and features [3], [4], and graph alignment [5].

Among alternative similarity measures, in this paper, we focus on graph edit distance (GED) [6]. GED is considered as one the most flexible and versatile graph matching models available, and has been widely used in image analysis, handwritten document analysis, biometric, bio/cheminformatics, knowledge and process management, malware detection, and other applications [7].

The associate editor coordinating the review of this manuscript and approving it for publication was Chao Tan.

The GED between two graphs is the minimum number of edit operations to transform one graph to the other, where an edit operation is insertion, deletion, or substitution of a single vertex or edge. The advantage of GED is that it is very sensitive to differences between graphs. Therefore, it is mainly used for rather small graphs, where it is crucial to capture the difference very precisely [7].

GED computation for a pair of graphs is a process to find an optimal vertex mapping between the pair that incurs the minimum number of edit operations to make the pair isomorphic. Since each vertex in one graph can be mapped to any vertex in the other graph, the number of all possible vertex mappings is exponential to the number of vertices in the involved graphs. In fact, the problem of GED computation has been proved to be NP-hard [10] (see Section II-A for the details of the vertex mapping and GED problem).

Existing algorithms implicitly organize all the possible vertex mappings into a prefix-shared search tree, where each leaf node of the tree represents a vertex mapping. Most existing

TABLE 1. **Comparisons of** HGED **with the existing parallel algorithms.**

Algorithms	**Search strategy**	**Load balancing**	**Upper bounds**	
			Detection	Tightness
PDF [8]	Depth-first search	Static + dynamic load balancing	Fast and frequent	Loose
PGED [9]	Best-first search (A* algorithm)	Static load balancing	Slow and infrequent	Tight
HGED	Hybrid two-phase search (Section III-B)	Static + dynamic load balancing	Fast and frequent	Tight

algorithms adopt the A* search to explore the search tree (e.g. [11]–[14]), while a few algorithms traverse the search tree in a depth-first manner (e.g., [15], [16]). The main focus of existing algorithms is on reducing the search space by identifying and pruning subtrees whose leaves cannot be an optimal vertex mapping (see Section II-B for the details of the search tree and search algorithms).

To the best of our knowledge, only two parallel GED computation algorithms, PGED [9] and PDFS [8], have been proposed. These algorithms parallelize GED computation by dividing the search tree into disjoint subtrees that cover all leaf nodes, assigning the subtrees into different threads, and simultaneously exploring the subtrees. They share the strategy to divide the search tree and take similar approaches for the subtree assignment. The subtree assignment is also referred to static load balancing [9]. In contrast to the subtree assignment, they use different search strategies for subtree traversals. PGED simultaneously traverses the subtrees in a best-first (i.e., A*) fashion, while PDFS performs parallel depth-first traversals on the subtrees. These algorithms utilize GED upper bounds, which are obtained whenever a leaf node is found, to reduce the search space. The tighter an upper bound is, the more the search space is reduced. PDFS quickly finds upper bounds, but it hardly reduces the search space because the found upper bounds tend to be very loose. On the contrary, PGED can find tighter upper bounds. However, PGED cannot effectively use upper bounds in reducing search space because it very slowly produces upper bounds. We will present how a GED upper bound is used to save computation in both search strategies, and discuss in detail the limitations of existing solutions in Section III-A.

To address the limitations of the existing solutions, in this paper, we aim at designing a search strategy that quickly finds a tight lower bound. To this end, we propose a novel hybrid search scheme that combines the depth-first and best-first search paradigms. Based on the proposed scheme, we develop an efficient parallel GED computation algorithm HGED, which stands for **H**ybrid search for parallel **GED** computation. In addition to the hybrid search strategy, HGED is also equipped with a dynamic load balancing to maximize thread utility. If each thread cannot run independently, the overall performance significantly degrades due to the synchronization overhead. For this reason, PGED do not use a dynamic load balancing. PDFS uses a simple dynamic load balancing scheme, but their scheme requires a running thread to be synchronized for redistributing workload. The load balancing scheme proposed in this paper ensures that only idle threads will wait for workload for a short period and running threads do not need to be synchronized and never wait for other threads. The comparison of HGED with the existing parallel algorithms is shown in Table 1.

In summary, the following are the main contributions of this paper.

- We propose a novel search strategy that combines the depth-first and best-first search paradigms aiming at quickly finding a tight GED upper bound.
- We develop a dynamic load balancing technique to maximize thread utility. Our load balancing technique ensure that no running thread wait for synchronization.
- We integrate the proposed search strategy and load balancing technique into a parallel GED computation algorithm HGED and implement the algorithm.
- We conduct extensive experiments on real datasets and show that HGED significantly outperform the state-of-the art algorithms.

The rest of the paper is organized as follows: Section II presents preliminaries including the problem formulation and a brief survey of prior work. Section III describes our parallel GED computation algorithm HGED focusing on a novel hybrid two-phase search strategy. Section IV develops a dynamic load balancing technique that aim at maximizing thread utility. Section V reports experimental results and Section VI lists related work. Section VII discusses the findings and limitations, and Section VIII concludes the paper.

II. PRELIMINARIES

A. PROBLEM FORMULATION

In this paper, we focus on undirected and labeled simple graphs, though the proposed technique can be easily extended other types of graphs. An undirected and labeled simple graph g is a triple $(V(g),\ E(g),\ l)$, where $V(g)$ is a set of vertices, $E(g) \subseteq V(g) \times V(g)$ is a set of edges, and $l : V(g) \cup (V(g) \times V(g)) \rightarrow \Sigma$ is a labeling function that maps vertices and edges to labels, where Σ is the label set of vertices and edges. $l(v)$ and $l(u, v)$ respectively denote the label of a vertex v and the label of an edge (u, v). If there is no edge between u and v, $l(u, v)$ returns a unique value λ distinguished from all other labels. We also define a blank vertex ε such that $l(\varepsilon) = l(\varepsilon, v) = l(u, \varepsilon) = \lambda$. There are no self-loops nor more than one edge between two vertices. For simplicity, in the rest of the paper, graph denotes undirected and labeled simple graph.

Graph edit distance, which is used to measure the similarity between graphs in this paper, is defined as follows.

Definition 1 (Graph Edit Distance): The graph edit distance (GED) between two graphs g_1 and g_2, which is denoted by $\mathsf{ged}(g_1, g_2)$, is the minimum number of edit operations that transform g_1 into g_2, where an edit operation is one of the following:

1) insertion of an isolated labeled vertex
2) deletion of an isolated labeled vertex
3) substitution of the label (i.e., relabeling) of a vertex
4) insertion of a labeled edge
5) deletion of a labeled edge
6) substitution of the label (i.e., relabeling) of an edge.

Example 1: Consider two graphs g_1 and g_2 in Figure 1. The following three edit operations on g_1 transform g_1 into g_2: insertion of an edge between u_2 and u_3, deletion of the edge between u_3 and u_4, and substitution of the label of u_4 (from D to A). Therefore, $\mathsf{ged}(g_1, g_2) = 3$.

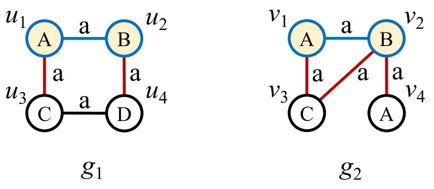


FIGURE 1. Example graphs.

To formulate the problem of GED computation, we first introduce a vertex mapping and edit cost in Definition 2 and Definition 3.

Definition 2 (Vertex Mapping): A *vertex mapping* between two graphs g_1 and g_2 is a bijection of $V(g_1)$ onto $V(g_2)$.[1] A vertex mapping is represented by an ordered set of mapped vertex pairs, where the order is imposed by a pre-defined ordering of $V(g_2)$.

Example 2: In Figure 1, consider the vertex ordering of g_2 is (v_1, v_2, v_3, v_4). $m = \{u_1 \mapsto v_1, u_2 \mapsto v_2, u_3 \mapsto v_3, u_4 \mapsto v_4\}$ is a vertex mapping between g_1 and g_2.

Given a vertex mapping m, g_1 can be transformed into g_2 by abiding by m as follows. For each mapped vertex pair $u \mapsto v \in m$, we make u and v identical in terms of the labels of the vertices and the labels and connectivity of their adjacent edges. The number of edit operations required in this transformation is called the *edit cost* of m, which is formally stated in Definition 3.

Definition 3 (Edit Cost): Let $u \mapsto v$ be the last mapped vertex pair in m, and $m' = m - \{u \mapsto v\}$. The edit cost of m is defined as:

$$\mathsf{ec}(m) = \mathsf{ec}(m') + \mathsf{d}[l(u), l(v)] + \sum_{u' \mapsto v' \in m'} \mathsf{d}[l(u, u'), l(v, v')],$$

where $\mathsf{ec}(\emptyset) = 0$ and $\mathsf{d}[x, y] = \begin{cases} 0, & \text{if } x = y \\ 1, & \text{otherwise.} \end{cases}$

The problem of GED computation is defined as follows.

[1]If $|V(g_1)| \neq |V(g_2)|$, $||(V(g_1)| - |V(g_2)||$ copies of a blank vertex ε are added into $V(g_1)$ or $V(g_2)$ to make $|V(g_1)| = |V(g_2)|$. For the ease of presentation, we assume $|V(g_1)| = |V(g_2)|$ in the remainder of the paper.

Definition 4 (GED Computation): Given two graphs g_1 and g_2, let $f(g_1, g_2)$ denote the bijection of $|V(g_1)|$ onto $|V(g_2)|$. The graph edit distance between g_1 and g_2 is computed as:

$$\mathsf{ged}(g_1, g_2) = \min_{m \in f(g_1, g_2)} \mathsf{ec}(m).$$

B. GED COMPUTATION ALGORITHM

This subsection provides a general description of existing serial GED computation algorithms. As stated in Definition 4, GED computation is a process to find a vertex mapping having a minimum edit cost among all possible vertex mappings between g_1 and g_2. To avoid redundant edit cost computation among vertex mappings that shares a prefix, all possible vertex mappings can be organized into a prefix tree, which is called a *search tree*.

An example search tree for the graphs in Figure 1 is depicted in Figure 2. In this example, the pre-defined vertex ordering of g_2 is (v_1, v_2, v_3, v_4). Each intermediate node n represents a *partial mapping*, which is a shared prefix of the vertex mappings in the leaves of the subtree rooted by n. Let the i^{th} vertex of g_2 be v. A tree node containing a vertex u of g_1 at level i represents a mapping $m_p \cup \{u \mapsto v\}$, where m_p is the mapping of the parent node, and the mapping of the root is $\emptyset$. In Figure 2, for example, the node indicated by an arrow corresponds to a partial mapping $m = \{u_1 \mapsto v_1, u_2 \mapsto v_2\}$. Since a partial mapping uniquely identifies a node in the search tree, in this paper, a partial mapping is interchangeably used with the corresponding tree node if clear from the context.

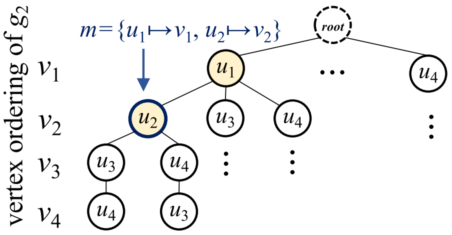


FIGURE 2. Search tree for graphs in Figure 1.

Definition 5 (Lower Bound of a Partial Mapping): The *lower bound* of a partial mapping m, denoted by $\mathsf{lb}(m)$, is a lower limit of the edit costs of the vertex mappings in the leaves of the subtree rooted by m.

Given a GED threshold τ, the subtree rooted by m is pruned if $\mathsf{lb}(m) > \tau$. To compute $\mathsf{lb}(m)$, each graph g participating in m is divided into the following three parts:

- The *mapped subgraph* of g, which is denoted by $g|_m$, is an induced subgraph of g defined by the vertices of g participated in m.
- The *unmapped subgraph* of g, which is denoted by $g \backslash g|_m$, is an induced subgraph of g defined by the vertices in $V(g) \backslash V(g|_m)$.
- The *bridges* are edges connecting $g|_m$ to $g \backslash g|_m$.

Then, $\mathsf{lb}(m)$ is computed as the sum of

1) the edit cost required between $g_1|_m$ and $g_2|_m$, which is computed as $\mathsf{ec}(m)$ in Definition 3;
2) a lower bound of the GED between $g_1 \backslash g_1|_m$ and $g_2 \backslash g_2|_m$, which is computed using the *label set-based lower bound* in Definition 6;
3) and a lower bound of the number of edit operations required to make the bridges of g_1 and g_2 identical, which is computed using the *bridge lower bound* in Definition 7.

Definition 6 (Label Set-Based Lower Bound [17]): The label set-based lower bound between two graphs r and s is defined as:

$$\mathsf{lb}_\mathsf{L}(r, s) = \Gamma(L_V(r), L_V(s)) + \Gamma(L_E(r), L_E(s)),$$

where $L_V(g)$ and $L_E(g)$ denotes the label multisets of vertices and edges of a graph g, respectively, and $\Gamma(A, B) = \max(|A|, |B|) - |A \cap B|$.

Definition 7 (Bridge Lower Bound [13]): Given a partial mapping m, the number of edit operations required in the bridges are at least

$$\mathcal{B}(m) = \sum_{u \to v \in m} \Gamma(L^m_{br}(u), L^m_{br}(v)),$$

where $L^m_{br}(w)$ denotes the label multiset of the bridges connected to a vertex w.

In summary, the lower bound of m is computed as

$$\mathsf{lb}(m) = \mathsf{ec}(m) + \mathcal{B}(m) + \mathsf{lb}_\mathsf{L}(g_1 \backslash g_1|_m, g_2 \backslash g_2|_m). \quad (1)$$

Example 3: Consider a partial mapping $m = \{u_1 \mapsto v_1, u_2 \mapsto v_2\}$ between the two graphs in Figure 1. The mapped subgraphs, unmapped subgraphs, and bridges of the graphs are depicted in blue, black, and red lines in the figure, respectively. The edit cost between $g_1|_m$ and $g_2|_m$ is 0 since $\mathsf{ec}(m) = 0$. There is one difference between the label multiset of vertices in $g_1 \backslash g_1|_m$, which is $\{C, D\}$, and that in $g_2 \backslash g_2|_m$, which is $\{C, A\}$. There is also one difference in the edge labels of the unmapped subgraphs, because the label multiset of the edges in $g_1 \backslash g_1|_m$ is $\{a\}$ and that in $g_2 \backslash g_2|_m$ is $\emptyset$. Therefore, $\mathsf{lb}_\mathsf{L}(g_1 \backslash g_1|_m, g_2 \backslash g_2|_m) = 2$. The bridge lower bound $\mathcal{B}(m)$ is 1 because the bridge label difference between u_1 and v_1 is 0 and that between u_2 and v_2 is 1. Therefore, $\mathsf{lb}(m) = 3$.

Existing solutions traverse the search tree in either a best-first (i.e., A*) or a depth-first manner. Algorithm 1 outlines a unified GED computation framework that can be used with both A* and depth-first search strategies. It first determines the order of vertices in g_2 (Line 1). A common intuition behind the vertex ordering is that infrequent vertices are matched first while preserving the connectivity [13], [14]. After determining the vertex ordering of g_2, the algorithm pushes the initial state, i.e., an empty mapping, which corresponds to the root node of the search tree, into the queue (Lines 2). It also initializes a GED upper bound ub, which denotes the lowest edit cost found so far (Line 3). In the main loop, it pops a mapping m from the queue (Line 5). If the mapping m popped from the queue is a *full mapping* (i.e., m contains all vertices of g_1 and g_2), it updates ub using the edit cost of m (Line 6). Note that $\mathsf{lb}(m) = \mathsf{ec}(m)$ when m contains all vertices of g_1 and g_2. If m is a partial mapping and its lower bound is less than ub (Line 7), it expands the search tree by mapping the next unmapped vertex v in g_2 (Line 6) to each unmapped vertex u in g_1 (Line 7). It pushes each expanded tree node m_c into the queue if the lower bound of m_c is less than ub (Lines 8–10). If the queue is empty, the algorithm returns ub, which is the lowest edit cost of all possible mappings between g_1 and g_2 (Line 12).

Algorithm 1: $\mathsf{GED}(g_1, g_2)$

```
input  : $g_1$ and $g_2$ are graphs.
output: $\mathsf{ged}(g_1, g_2)$
1  $\mathcal{O} \leftarrow$ vertex ordering of $V(g_2)$;
2  initialize a priority queue $Q$ with an empty mapping;
3  $ub \leftarrow \infty$;
4  while $Q \neq \emptyset$ do
5      $m \leftarrow Q.\mathsf{pop}()$;
6      if $|m| = |V(g_1)| \wedge \mathsf{lb}(m) < ub$ then $ub \leftarrow \mathsf{lb}(m)$;
7      if $\mathsf{lb}(m) < ub$ then
8          $v \leftarrow$ next unmapped vertex in $V(g_2)$ as per $\mathcal{O}$;
9          foreach $u \in V(g_1)$ s.t. $v \notin m$ do
10             $m_c \leftarrow m \cup \{u \mapsto v\}$;
11             if $\mathsf{lb}(m_c) < ub$ then $Q.\mathsf{push}(m_c)$;
12 return $ub$;
```

The search strategy (i.e. either A* or depth-first) can be determined by using a different priority of a mapping in the queue. To use A* search, the algorithm pops from the queue a mapping that has the minimum lower bound. If there is a tie, a mapping at a larger level is preferred. We remark that if the algorithm uses the A* search, it can be terminated as soon as it finds a full mapping. Because it pops from the queue a mapping having the smallest lower bound, any mapping remaining in the queue cannot have a less edit cost.

If a mapping having the largest size is popped from the queue, the algorithm traverses the search tree in a depth-first manner. Note that among all mappings in the queue, a mapping having the largest size is at the deepest level of the search tree. If there is a tie, a mapping having a smaller lower bound is preferred. In the depth-first search case, the priority queue plays a role of a stack.

C. PARALLEL GED COMPUTATION

This subsection briefly introduces the existing parallel GED computation algorithms, PDFS [8] and PGED [9]. Existing approaches perform the following steps for parallel GED computations.

Step 1: The main thread takes several iterations of the main loop in Algorithm 1 using A* until the queue contains enough tree nodes (i.e., partial mappings). For example, PGED runs this step until the size of the queue reaches to $n \times \beta$, where n is the number of threads and β is a tunable parameter whose default value is set to 20.

Step 2: The main thread distributes the tree nodes to worker threads. Existing solutions take a greedy approach for assigning tree nodes to each thread. For example, PDFS assigns the tree nodes in the queue to each thread in the sorted order of their lower bounds. PGED uses a similar approach for the assignment.

Step 3: Each worker thread traverses the assigned subtrees using its own queue. PDFS traverses subtrees in a depth-first manner, while PGED uses A* search in each thread. In this step, a GED upper bound shared by all threads is used to maintain the lowest edit cost found so far. The upper bound is used to reduce the search space as follows. When a thread traverses a tree node m, the subtree rooted by the node is pruned if $\mathsf{lb}(m)$ is not less than the upper bound.

Step 4: To ensure no thread remains idle, a dynamic load balancing scheme can be introduced in this step. In PDFS, if a thread becomes idle, the thread having the heaviest workload will be in charge of giving some workload to the idle thread. PGED does not use a dynamic load balancing.

III. HYBRID SEARCH TECHNIQUE FOR PARALLEL GED COMPUTATION

This section proposes a novel hybrid search strategy that combines the A* and depth-first search paradigms. Section III-A first presents the motivation of our work by analyzing existing search techniques focusing on the GED upper bound. Then, Section III-B develops our hybrid search algorithm for parallel GED computation.

A. MOTIVATION OF OUR WORK

As described in Algorithm 1 and in Section II-C, the shared GED upper bound plays an important role to reduce the search space. In case of the depth-first search in PDFS, GED upper bounds are frequently found whenever a thread reaches a leaf node. If the A* search in PGED is used, however, each thread finds only one upper bound, because the thread pops all mappings in the queue and terminates its search as soon as it meets a leaf node. The utilization of the upper bound in the depth-first search is rather straightforward. In case of the A* search, however, it is complicated how the upper bound affects the search space. Hence, before discussing the limitations of existing solutions and the motivation of our work, we analyze the effect of the upper bound in case of the A* search.

Figure 3 illustrates a situation where there are three worker threads, t_1, t_2, and t_3, and t_2 has found an upper bound ub_2 while t_1 and t_3 are still traversing their search spaces. Thus, ub_2 is the GED upper bound shared by t_1 and t_3. Suppose the upper bounds of t_1 and t_3 will be ub_1 and ub_3, respectively, and $ub_1 < ub_2 < ub_3$. As shown in the figure, t_3 is terminated without producing an upper bound ub_3, as soon as a mapping m such that $\mathsf{lb}(m) \geq ub_2$ is popped from its queue. This is

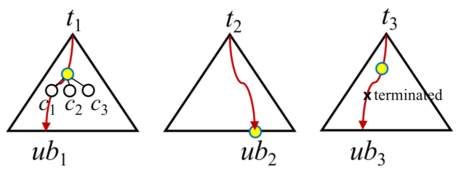


FIGURE 3. Example of parallel A* search: red lines denote optimal search paths; yellow circles denote currently traversed nodes; c_i denotes each child of the current node in t_1; and ub_i denotes the upper bound of each thread.

because the lower bounds of all the mappings remaining in the queue cannot be less than $\mathsf{lb}(m)$ in the A* search.

Algorithm 2: $\mathsf{lb}(m, ub)$
```
1 $lb \leftarrow \mathsf{ec}(m)$;
2 if $lb < ub$ then $lb \leftarrow lb + \mathcal{B}(m)$;
3 if $lb < ub$ then $lb \leftarrow lb + \mathsf{lb}_\mathsf{L}(g_1 \backslash g_1|_m, g_2 \backslash g_2|_m)$;
4 return $lb$;
```

For t_1, however, all the mappings popped from its queue have lower bounds less than the shared upper bound ub_2, since any mapping whose lower bound is greater than ub_1 cannot be popped from the queue in the A* search. Therefore, the search space cannot be reduced by ub_2. Nevertheless, the shared upper bound ub_2 can be effectively used to reduce the computational overhead of t_1 as follows. When t_1 expands its child nodes c_1, c_2, and c_3, it computes the lower bounds of c_i's using Equation 1 in Section II-B. Given an upper bound ub, Equation 1 can be incrementally computed as shown in Algorithm 2. Because c_1 is on the optimal search path, its lower bound is obviously less than the shared upper bound ub_2. Hence, it is required to compute ec, $\mathcal{B}$, and lb_L of c_1 in Algorithm 2, and computation for c_1 cannot be saved. Using ub_2, however, lower bound computation for c_2 and c_3 can be saved depending on their ec and $\mathcal{B}$ values. If $\mathsf{ec}(c_2) \geq ub_2$, for example, it is not needed to compute $\mathcal{B}$ and lb_L of c_2. We remark that A* computes lower bounds of a large amount of tree nodes that will be never traversed. Hence, an upper bound can significantly save computation when using the A* search strategy.

Now, we discuss the limitations of the existing parallel GED computation techniques. The depth-first search strategy in PDFS quickly produces an upper bound, but the found upper bound tends to be very loose. Consider PDFS first traverses a path in a subtree whose leaf nodes have large edit costs. It cannot escape from the subtree until it traverses all the nodes in the subtree. Although it can produce an upper bound quickly, we hardly expect that the upper bound is tight considering the majority of leaves have edit costs much larger than the actual GED.

We can expect that the A* search strategy in PGED produces a tighter upper bound. However, it can produce an upper bound after a thread completely traverses its search space. Therefore, an upper bound tends to be found too late to be effectively used in reducing the search overheads of other threads. If an upper bound happens to be found early, then this approach should suffer from poor thread utilization because the thread produced the upper bound becomes idle. We will empirically show these limitations of existing solutions in Section V-B2.

To address the limitations of existing search strategies, we aim at developing a novel search strategy that can quickly find a tighter upper bound. The following section presents the details of our search strategy.

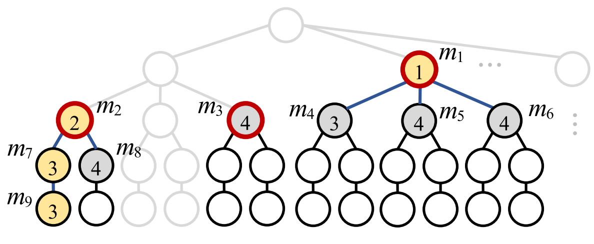
B. HGED: HYBRID PARALLEL SEARCH ALGORITHM

This subsection presents our hybrid parallel search algorithm HGED. HGED follows the first two steps of existing techniques (Section II-C), which are summarized as follows. HGED first performs the A* search on the search tree until the priority queue contains enough nodes (i.e., partial mappings). Then, it distributes the nodes in the queue into worker threads. HGED adopts the greedy assignment of PGED for the distribution.

Now, we discuss how a thread traverses the subtrees rooted by the assigned nodes. The design objective of our search strategy is twofold: (i) a thread reaches to a leaf node as fast as the depth-first search and (ii) an upper bound found in a leaf node is a tight one. To achieve the goal, we develop a two-phase search scheme. The proposed scheme traverses the search tree as follows. In the first phase, it selects a node having the minimum lower bound from the queue just like the A* search. In the second phase, it travels as deep as possible to a leaf node. In each iteration of the second phase, it increases the level by one and selects a node having the minimum lower bound at the level. The second phase continues until either a leaf node is found or all the nodes in the current level are pruned. Once the second phase is finished, it switches to the first phase to select a node instead of backtracking. The intuition behind the two-phase search strategy is that we increase the chance to obtain a tighter upper bound by starting the traversal with the most promising node, while we produce an upper bound quickly by traversing a node in a higher level of the tree in each step of the traversal. We also improve the tightness of an upper bound by selecting the most promising node at the current level when we traverse to a leaf.

The following example demonstrates how our two-phase search strategy traverses the search tree.

Example 4: Figure 4 shows the search space of a thread. In the figure, a number inside a node denotes the lower bound of the node, which is arbitrarily chosen for illustration purpose. Consider three mappings m_1, m_2, and m_3 are initially assigned to the thread. The thread first takes m_1, which has the minimum lower bound among the three mappings, and expands the tree by generating child mappings m_4, m_5, and m_6. Then, the thread increases the level by one, chooses m_2, which has the lowest lower bound at the current level, and generates child mappings m_7 and m_8. After increasing the level by one, in the same way, it chooses m_7 whose lower bound is minimum among the nodes available at the current level, and generates m_9. It finally reaches to a leaf node m_9 and finds a GED upper bound 3. After the thread finds a leaf node, it restarts the traversal by selecting a node having the lowest lower bound in the queue, which is m_4.



⭕ initially assigned nodes	🟡 traversed nodes
⚪ nodes remaining in the queue	○ unseen nodes

FIGURE 4. A running example of the hybrid search.

Before we present the details of our two-phase search algorithm, we describe the implementation of a priority queue for the algorithm. To support the two-phase search scheme, we implement a priority queue with multiple min-heaps. We maintain a heap called a *local heap* for each level to find the minimum lower bound mapping in each level, and use another heap called a *global heap* to keep track of a mapping having the minimum lower bound in the entire search space assigned to a thread. To find such a mapping, it is enough for the global heap to contains a copy of the minimum lower bound mapping of each local heap. Figure 5 shows an example priority queue containing the grey-colored nodes in Figure 4. We note that the level of the root node is 0.

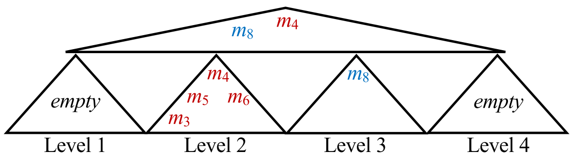


FIGURE 5. Priority queue for Figure 4.

To implement push and pop operations of a priority queue, heaps in the queue support the following operations.

- insert(m) inserts a mapping into the heap.
- replace(m) replaces the mapping whose size is $|m|$ by m, and reconstruct the heap. If there is no mapping whose size is $|m|$, it inserts m into the heap. This operation assumes that there is at most one mapping whose size is $|m|$ in the heap.
- get_min() returns a mapping having the minimum lower bound.
- extract_min() returns a mapping having the minimum lower bound, and delete the mapping from the heap.

Algorithm 3 shows the push operation. Given a mapping m, the level lv of m is the number of mapped pairs of vertices in m, since a pair of vertices mapped at each level (Line 1). Thus, the algorithm pushes m into heap[lv], which denotes the local heap for the level lv (Line 2). If m has the minimum

Algorithm 3: push(m)

```
input  : $m$ is a partial mapping.
output: void
1  $lv = |m|$;
2  heap[$lv$].insert($m$);
3  if heap[$lv$].top() $= m$ then heap[global].replace($m$);
```

lower bound in the local heap, it replaces, in the global heap, the mapping copied from this local heap by m (Line 3).

Algorithm 4: pop(lv)

```
input  : $lv$ is the level of the search tree.
output: void
1  if heap[$lv$] $= \emptyset$ then return nil;
2  $m \leftarrow$ heap[$lv$].extract_min();
3  if $lv =$ global then
4      heap[$|m|$].extract_min(); //delete $m$
5      $m' \leftarrow$ heap[$|m|$].get_min();
6      if $m' \neq$ nil then heap[global].insert($m'$);
7  else
8      $m' \leftarrow$ heap[$lv$].get_min();
9      if $m' \neq$ nil then heap[global].replace($m'$);
10 return $m$;
```

The pop operation is outlined in Algorithm 4. Given a level lv, the algorithm basically extracts the minimum mapping from heap[lv] (Line 2), and returns the mapping (Line 10). If the mapping is extracted from the global heap (i.e., $lv =$ global, Line 3), it also removes the mapping from its local heap (Line 4), and inserts a copy of the minimum mapping of the local heap into the global heap (Lines 5–6). Otherwise, in the global heap, it replaces the extracted mapping by the minimum mapping in heap[lv] (Lines 7–9).

The following lemma states our priority queue implementation does not increase the time complexity compared with a conventional priority queue structure which is typically implemented using a single heap.

Lemma 1: The time complexity of a pop or push operation is $O(\log n)$, where n is the number of distinct mappings in the priority queue.

Proof: insert, replace, and extract_min operations of a heap requires $O(\log m)$ and get_min requires $O(1)$, where m is the size of the heap and $m \leq n$. Since pop and push perform a constant number of heap operations, the time complexity is $O(\log m) = O(\log n)$. □

Algorithm 5 encapsulates the proposed two-phase search. The algorithm first initializes the priority queue with the given workload (Lines 1–2). Then, it repeatedly pops from the queue a mapping m at the current level lv (Line 5), which is initially set to global (Line 3). If there is no mapping available at the current level or the lower bound of the mapping m is not less than the GED upper bound ub, the algorithm switches to

Algorithm 5: TwoPhaseSearch(W, ub)

```
input  : $W$ is the workload.
         $ub$ is the shared upper bound.
output: void
1  initialize an empty queue $Q$;
2  foreach $m \in W$ do $Q$.push($m$);
3  $lv \leftarrow$ global;
4  while $Q \neq \emptyset$ do
5      $m \leftarrow Q$.pop($lv$);
6      if $m =$ nil or lb($m$) $\geq ub$ then $lv \leftarrow$ global;
7      else if $m$ is a leaf node then
8          if lb($m$) $< ub$ then $ub \leftarrow$ lb($m$);
9          if $lv =$ global then return;
10         $lv \leftarrow$ global;
11     else
12         foreach child node $m_c$ of $m$ do
13             if lb($m_c$) $< ub$ then $Q$.push($m_c$);
14         $lv \leftarrow |m| + 1$;
```

the first phase by setting the level lv to global (Line 6). Otherwise, it handles the following two different cases. In case that m is a leaf node (Line 7), the algorithm synchronously updates the shared GED upper bound ub (Line 8). If m comes from the global heap (Line 9), the algorithm can immediately return (see Lemma 2 below). Otherwise, it switches to the first phase by setting the level lv to global (Line 10). In case that m is not a leaf (Line 11), the algorithm generates child nodes of m and push each child into the queue if its lower bound is less than ub. Then, the algorithm continues to search in the second phase by increasing the level by one (Line 14).

The two-phase search algorithm obviously traverses the search tree in a depth-first manner in that it always increases the level by one. The difference is that it does not backtrack and it can jump to different subtrees when it goes down to the tree. The two-phase search algorithm also inherits the A* algorithm as stated in the following lemma.

Lemma 2: TwoPhaseSearch finds a minimum edit cost in its search space as soon as it extracts a leaf node from the queue with the global level, i.e., the global heap.

Proof: Since the global heap contains the minimum mapping at each level, the mapping popped from the global mapping has the minimum lower bound among all expanded mappings. Therefore, no mapping in the queue have a chance to be extended to a mapping having a lower bound less than that of the mapping popped from the global heap. □

Algorithm 6 presents our parallel GED computation algorithm HGED that uses TwoPhaseSearch. The algorithm first divides the search tree into subtrees by running the serial A* algorithm (Line 1). If the queue size reaches to β times of the number of threads, it distributes the subtrees to n threads (Line 2). Like existing solutions, the nodes in the queue are sorted by their lower bounds and sequentially

```
Algorithm 6: HGED($g_1, g_2, n$)
input  : $g_1$ and $g_2$ are graphs; $n$ is the number of threads.
output: ged($g_1, g_2$)
1 run A* GED algorithm while $|Q| < \beta \times n$;
2 assign nodes in $Q$ to $W_1, \ldots, W_n$;
3 $ub \leftarrow \infty$; //sharedupperbound
4 for $i \leftarrow 1$ to $n$ do
5     spawn TwoPhaseSearch($W_i$, $ub$);
6 wait until all threads are terminated;
7 return $ub$;
```

assigned to the n threads. After the assignment, it initializes the shared GED upper bound (Line 3). Then, it spawns n threads with TwoPhaseSearch (Lines 4-5) and wait until all threads terminate their search (Line 6). It finally returns the upper bound, which is the lowest edit cost (i.e. GED) among all possible mappings. While simultaneously traversing the search tree, HGED performs a dynamic load balancing if a thread becomes idle. The details of the dynamic load balancing will be presented in the following section.

IV. DYNAMIC LOAD BALANCING

This section presents the dynamic load balancing algorithm of HGED. The existing solution PDFS uses a simple dynamic load balancing scheme that the heaviest thread redistributes its workload to an idle thread. PDFS does not provide the details of its load balancing scheme, but it requires synchronization among running threads to elect a thread having the heaviest workload.

In this paper, we develop a novel dynamic load balancing scheme, which guarantees that no running thread waits for synchronization. To this end, a shared array consisting of n elements is maintained, where n is the number of threads. Each element in the array has two slots named *wsize* and *requester*. Then, the shared array is used for the load balancing as follows.

- **Running thread:** In every iteration of TwoPhaseSearch (i.e., at the beginning of the **while** loop in Line 4 of Algorithm 5), each running thread t_i asynchronously accesses the i^{th} element of the shared array for (i) updating wsize with the size of its current search space and (2) checking if there is an idle thread contained in requester. If there is a waiting idle thread, t_i shares its workload to the idle thread, set requester to nil, and continues its search.
- **Idle thread:** Each idle thread asynchronously reads all the wsize's from the shared array and sorts the thread ids in a decreasing order of their wsize's, where the id of the i^{th} thread is i. Then, the idle thread tries to lock requester from the heaviest thread to the lightest one among threads having at least certain amount of workload. If it obtains a lock, then it waits until the requester slot becomes nil. After waking up, the idle thread unlocks the requester and restarts its search. If it fails to lock any requester, it repeats the process until either it obtains a lock or all threads become idle.

Example 5: Figure 6 shows an example of matching idle threads with running threads. In the figure, there are two idle threads t_x and t_y among n threads. Each running threads t_i (e.g., t_1, t_2, t_3, and t_n in the figure) asynchronously updates the wsize slot and checks the requester slot of the i^{th} element of the shared array. The idle threads find target threads as follows: ① The idle threads simultaneously read wsize's and sort the thread ids by their wsize's. ② Both of the idle threads try to lock the requester slot of the heaviest thread t_2 (i.e., the second element of the shared array). In this example, t_x happens to obtain the lock while t_y fails to lock it. Hence, t_x writes its thread id into the requester. ③ t_y has a lock on the requester slot of the second heaviest thread t_n and writes its thread id into the requester slot of t_n.

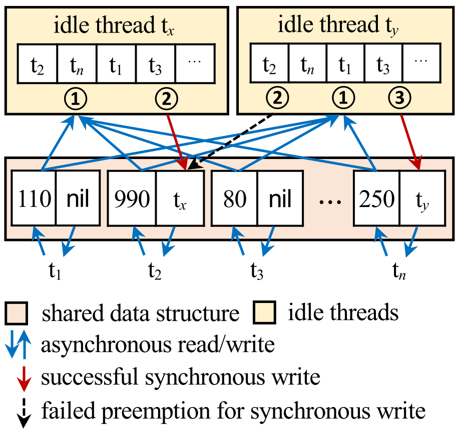


FIGURE 6. Example of thread matching in our dynamic load balancing scheme.

Our scheme may not correctly reflect the workload sizes since wsize's are read and updated asynchronously. That is, when an idle thread selects the heaviest running thread, the selected thread may not be the heaviest as the workloads of running threads change. Nonetheless, it does not affect the correctness of our algorithm and the size difference is not significant. Therefore, our scheme focuses on reducing the synchronization overhead.

Share in Algorithm 7 outlines the load balancing algorithm for a running thread t_i. Share is called at the beginning of every iteration of the **while** loop of Algorithm 5. It first updates the size of the search space of the thread t_i (Line 1). If there exists a waiting thread t_j for load balancing (Lines 2–3), the algorithm shares the workload of t_i with t_j (Lines 4–13). For each level lv of the search tree, it retrieves mappings from the heap of t_i at the level, which is denoted by t_i.heap[lv], and distributes the mappings into the two arrays H_{t_i} and H_{t_j} (Lines 6–9). Then, it replaces the heaps of t_i and t_j at the level lv by H_{t_i} and H_{t_j} (Lines 10–11). Note that it is not required to construct the heaps for H_{t_i} and H_{t_j}

```
Algorithm 7: Share(SA, $t_i$) //running thread
input : SA is the shared array; $t_i$ is a running thread.
output: void
1  SA[$i$].wsize ← the size of the workload of $t_i$;
2  if SA[$i$].requester ≠ nil then
3      $t_j$ ← SA[$i$].requester;
4      foreach search tree level $lv$ do
5          initialize two empty array $H_{t_i}$ and $H_{t_j}$;
6          while $t_i$.heap[$lv$] ≠ ∅ do
7              $H_{t_i}$.append($t_i$.heap[$lv$].extract_min());
8              if $t_i$.heap[$lv$] ≠ ∅ then
9                  $H_{t_j}$.append($t_i$.heap[$lv$].extract_min());
10         replace $t_i$.heap[$lv$] with $H_{t_i}$;
11         replace $t_j$.heap[$lv$] with $H_{t_j}$;
12     reconstruct $t_i$.heap[global] and $t_j$.heap[global];
13     SA[$i$].requester ← nil;
       //wake up the requester
```

because mappings in H_{t_i} and H_{t_j} are already sorted by their lower bound. After the algorithm distributes mappings in all levels, it reconstructs the global heaps for t_i and t_j (Line 12). Finally, it wakes up the waiting thread t_j (Line 13).

```
Algorithm 8: Request(SA, $t_j$) //idle thread
input : SA is the shared array; $t_j$ is an idle thread.
output: true if successful, false otherwise.
1  SA[$j$].wsize ← 0;
2  SA[$j$].requester ← nil;
   //wake up a requester if any
3  $T$ ← an array of thread ids sorted using SA.wsize;
4  foreach $i \in T$ s.t. SA[$i$].wsize > 0 do
5      if try_lock(SA[$i$].requester) then
6          SA[$i$].requester ← $t_j$;
7          wait_until(SA[$i$].requester = nil);
8          unlock(SA[$i$].requester);
9          return $t.Q \neq \emptyset$;
10 return false;
```

Request in Algorithm 8 shows the load balancing algorithm for an idle thread t_j. When the queue becomes empty in the **while** loop of Algorithm 5, the Request function is called instead of terminating the search. Request first sets the wsize of the idle thread t_j to zero, and wakes up a requester thread if any (Lines 1–2). Then, it sorts thread ids using their workload sizes (Line 3), and investigates threads from the heaviest to the lightest one (the **foreach** loop in Line 4). The algorithm tries to lock t_i (Line 5) and if it acquires the lock, it sets the requester of t_i (Line 6) and waits until t_i shares its workload (Line 7). Then, it unlocks t_i and return (Lines 8–9). If the algorithm fails to lock any running threads, it returns false (Lines 10). We note that the Request function is continuously called until either the queue of the calling thread is filled or entire threads become idle.

V. EXPERIMENTS

A. EXPERIMENTAL SETTINGS

We conducted experiments on two widely used datasets, AIDS and PubChem. AIDS is an antiviral screen compound dataset published by NCI/NIH (https://cactus.nci.nih.gov/download/nci/AIDS2DA99.sdz). It is a popular benchmark used in most graph search techniques. PubChem is a chemical compound dataset (https://pubchem.ncbi.nlm.nih.gov, Compound_000975001_001000000.sdf). It is a subset of chemical compounds published by the PubChem Project. Graphs in the PubChem dataset contain repeating substructures and have less size and label variations compared with the AIDS dataset.

TABLE 2. Statistics of datasets.

Dataset	$\|\mathcal{D}\|$	$\|V\|_{avg}$	$\|E\|_{avg}$	$\sigma_{\|V\|}$	$\sigma_{\|E\|}$	n_{vl}	n_{el}
AIDS	42,689	25.60	27.60	12.2	13.3	62	3
PubChem	22,794	48.11	50.56	9.4	9.9	10	3

Table 2 shows statistics of the AIDS and PubChem datasets. In the table, $|\mathcal{D}|$ is the number of graphs in each dataset, $|V|_{avg}$ and $|E|_{avg}$ is the average numbers of vertices and edges, $\sigma_{|V|}$ and $\sigma_{|E|}$ are the standard deviations of the numbers of vertices and edges, and n_{vl} and n_{el} are the numbers of distinct vertex and edge labels.

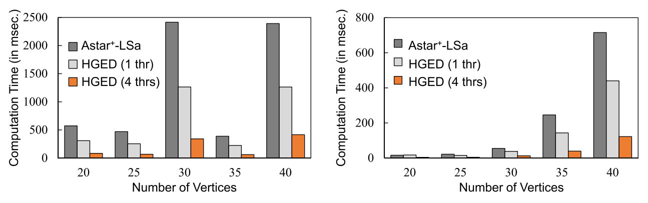
To conduct experiments, we generate 6 groups of graph pairs for the AIDS and PubChem datasets, respectively, as follows. For each group, we assign a number n of vertices and collect all graphs in the dataset whose vertex sizes are within the range of $[n-2, n+2]$. From the collected graphs, we then randomly select 30 pairs whose GED is within the range of [6, 11]. The numbers of vertices assigned to these groups are 20, 25, 30, 35, 40, and 45, respectively. In the experiments, we set the default value of the parameter β to 20 as suggested in [9]. If not stated otherwise, we used 4 threads for parallel computation. The average GED computation time of 30 pairs are reported in the experiments.

We implemented HGED in C++, and compiled it using GCC with the -O3 flag. In the implementation of HGED, we used the techniques for incremental lower bound computation and blank vertex removal in a vertex mapping proposed in [14]. All queries were evaluated on a machine with an Intel core i7 and 32GB RAM running a 64-bit Ubuntu OS.

B. EXPERIMENTAL RESULTS

1) COMPARISON WITH THE SERIAL ALGORITHM AStar$^+$-LSa

We first compared HGED with the state-of-the art serial GED computation algorithm AStar$^+$-LSa [14]. For the comparison, we used HGED with a single thread and HGED with 4 threads. Figure 7 shows the results.



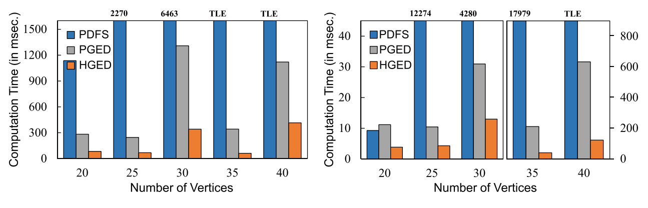
(a) AIDS (b) PubChem

FIGURE 7. **Comparisons with** AStar$^+$-LSa**.**

As shown in the figure, HGED with 4 threads was 5 times faster than AStar$^+$-LSa on average. Interestingly, HGED with a single thread outperformed AStar$^+$-LSa by about 1.5 times. In a serial computation, the search space of the A* algorithm is provably minimum. Thus, HGED with a single thread cannot further reduce the search space. However, HGED can reduce the overhead of lower bound computation by quickly finding tighter GED upper bounds (refer to Section III-A to see how a GED upper bound can reduce the overhead of lower bound computation).

2) COMPARISON WITH PARALLEL ALGORITHMS PDFS AND PGED

We compared HGED with the state-of-the art parallel GED computation algorithms PDFS [8] and PGED [9]. As PDFS used an out-dated GED computation technique, we implemented PDFS based on DFS$^+$-LSa [14]. Figure 8 shows the experimental results.



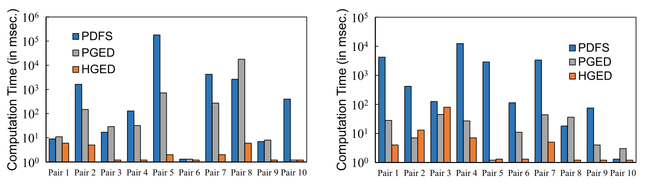
(a) AIDS (b) PubChem

FIGURE 8. **Comparisons with** PDFS **and** PGED**.**

As shown in the figure, HGED significantly and consistently outperformed existing parallel algorithms. It was about 3.5 times faster than PGED on both datasets. PDFS failed to complete GED computation for 30 pairs within the time limit (1 hour) for some groups. This is because the huge search space of the depth-first search paradigm.

The improvement of HGED can be explained by the proposed hybrid search algorithm **TwoPhaseSearch**. To see the efficiency of our hybrid search algorithm, we conducted the following experiments on 10 pairs of graphs randomly sampled from the group whose vertex size is 30. For each pair of graphs, we measured the elapsed time for finding the first upper bound equivalent to the GED of the pair.[2] Figure 9 shows the results. On the AIDS dataset in Figure 9(a), HGED found the upper bounds 700 times faster than PGED and and 7000 times faster than PDFS. Similar results were observed on the PubChem dataset as shown in Figure 9(b).

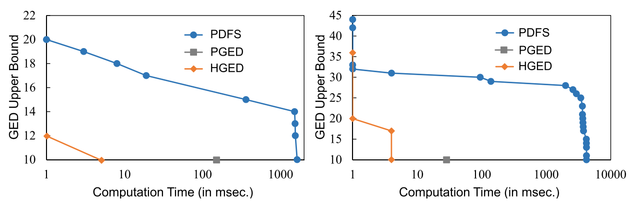
[2]We note that the time for finding the first upper bound equivalent to the GED is different from the GED computation time. Although we have found the upper bound equivalent to the GED, there will be remaining mappings whose lower bound is less than GED. GED computation algorithms should process all of these mappings.



(a) AIDS (b) PubChem

FIGURE 9. **Elapsed time for finding the first upper bound equivalent to the GED (y-axis is log-scaled).**

To see the change of upper bounds until the tightest one is found, we select a pair whose GED is 10 from each dataset, respectively. Figure 10 shows the results. On the AIDS dataset, the proposed approach could find the tightest upper bound within 10 milliseconds after finding an upper bound 12. On the PubChem dataset, our algorithm also found the tightest upper bound very quickly after finding a few upper bounds. The experimental results justify that our hybrid search algorithm can quickly find tighter lower bounds, which significantly reduces the search space in parallel GED computation. In serial computation, the tighter bounds also play an important role of reducing the overhead of lower bound computation as we demonstrated in Figure 7.



(a) AIDS (b) PubChem

FIGURE 10. **The change of upper bounds: Pair 2 for AIDS and Pair 1 for PubChem (x-axis is log-scaled).**

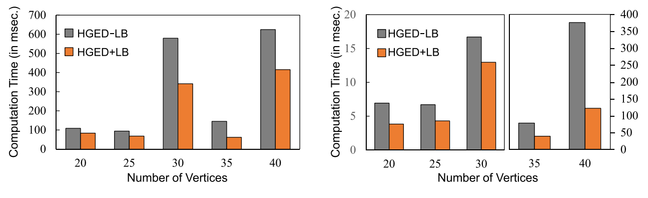
3) EVALUATION OF THE PROPOSED DYNAMIC LOAD BALANCING

We evaluated the effect of the proposed dynamic load balancing scheme, and reported the results in Figure 11. In the figure, HGED + LB and HGED-LB denote HGED with and without the load balancing technique, respectively.

On the AIDS dataset, we observed that HGED+LB consistently outperformed HGED-LB. The improvement was from 1.3 times to 2.3 times. Similar results were observed on the PubChem dataset, and the improvement was from 1.3 times to 3 times.

4) EVALUATION ON THE NUMBER OF THREADS

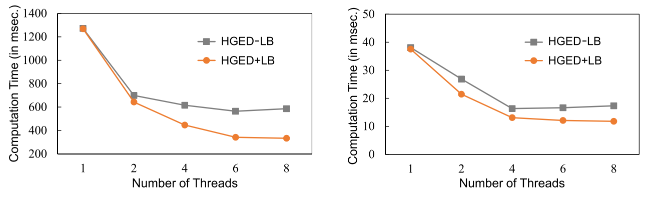
We vary the number of threads in the set {1, 2, 4, 6, 8}, and report the running times of HGED + LB and HGED-LB



(a) AIDS (b) PubChem

FIGURE 11. Evaluation of the dynamic load balancing.

in Figure 12. For the experiments, we used the group whose vertex size is 30 on each dataset.



(a) AIDS (b) PubChem

FIGURE 12. Effect of the number of threads.

From the results, we observed that the performance of HGED was greatly improved until $t = 4$, where t is the number of threads. If the load balancing scheme was not used, however, the performance was hardly improved for $t \geq 4$. This is because of the poor thread utilization. When we applied the load balancing scheme, the performance was gradually improved when t increases from 4 to 8. As noted in [9], this is mainly because of the overhead of resource allocation among threads and the cost for accessing and exchanging data. As the number of threads increases, those overhead and cost will also grow. Thus, the running time of HGED will not be reduced proportionally to the number of threads.

VI. RELATED WORK

The most widely used algorithm for GED computation is A*-GED [11]. Recently, BLP-GED [18], DF-GED [15], and CSI_GED [16], [19] have been proposed to improve the performance of GED computation. **BLP-GED** formulates the problem as a binary linear program, and it is faster and more memory-efficient than A*-GED. DF-GED traverses the search space in a depth first fashion. It has been found to much more memory-efficient than A*-GED. In contrast, CSI_GED proposed an edge-based depth-first search. It also has been found to be both much faster and more memory-efficient than A*-GED. Recently, AStar$^+$-LSa [14] has been proposed for speeding up GED computation. AStar$^+$-LSa reduces the search space by removing blank vertices from a vertex mapping and utilizing the bridge lower bound. It further speeds up GED computation by introducing an efficient lower bound computation technique. Two parallel GED computation algorithms, PGED [9] and PDFS [8], have been proposed. These algorithms aims at speeding up GED computation through parallel computing. The major difference is that PGED is based on AStar$^+$-LSa while PDFS is based on DFGED.

To solve the problem of graph similarity search, which is to find graphs similar to a given query using GED, efficient GED computation algorithms have been studied in a slightly different context. In the graph search problem, the focus is on finding pairs whose GED is within a given threshold rather than computing exact GEDs. Most of existing solutions for graph similarity search are based on the A*-GED algorithm. GSimSearch [12], [17] has suggested that the lower bound computation of A*-GED be improved by utilizing the label set differences. This approach is much faster than the bipartite heuristic used in A*-GED. Pars [20], [21] has proposed the extension-based verification which reduces the search space of GED computation. Inves [13] has introduced a lower bound estimation technique for bridges, which substantially reduces the search space. Inves and GSimSearch also exploited effective vertex orderings for improving the performance of GED computation.

VII. DISCUSSIONS

HGED aims to reduce the search space by quickly finding a tight upper bound. The proposed two-phase search algorithm used in HGED guarantees that it finds an upper bound within at most h iterations, where h is the height of the search tree. While traversing to a leaf node, the proposed search algorithm selects the best node at the current level. Once it meets a leaf node, the algorithm selects the best node from the entire search space, which is the start point of the next traversal to find another leaf. This property makes our algorithm mimic A* search strategy, and enables it to find a tight upper bound. We empirically observed from Section V that our algorithm can quickly find a tight upper bound and thus significantly reduce the search space. The experimental results justify the motivation of our work.

The main limitation of parallel GED computation techniques including the proposed one is that the speedup is not proportional to the number of threads. The dynamic load balancing scheme in HGED prevents running threads from waiting for load balancing to increase thread utilization. If there are a large number of threads, however, more idle threads compete for workloads of heavy threads, which results in poor idle thread utilization. We leave it as a future work to develop a load balancing technique that can significantly reduce the overhead of thread synchronization so as to maximize thread utilization.

VIII. CONCLUSION

In this paper, we study the problem of parallel GED computation. We propose a novel search algorithm HGED for simultaneously traversing the search tree. HGED can quickly find tighter GED upper bounds, which plays an important role of pruning a large amount of the search space. We also propose a dynamic load balancing scheme that aims at minimizing the synchronization overhead. In the proposed scheme, no running thread waits for synchronization. Experimental results demonstrate HGED significantly outperforms existing GED computation algorithms. HGED was from 4 to 7 times faster than the state-of-the art serial algorithm AStar^{+}-LSa, and from 3 to 6 times faster than the state-of-the art parallel algorithm PGED.

REFERENCES

[1] H. Shang, X. Lin, Y. Zhang, J. X. Yu, and W. Wang, "Connected substructure similarity search," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, 2010, pp. 903–914.

[2] H. Bunke and K. Shearer, "A graph distance metric based on the maximal common subgraph," *Pattern Recognit. Lett.*, vol. 19, nos. 3–4, pp. 255–259, Mar. 1998.

[3] S. Zhang, J. Yang, and W. Jin, "SAPPER: Subgraph indexing and approximate matching in large graphs," *Proc. VLDB Endowment*, vol. 3, nos. 1–2, pp. 1185–1194, Sep. 2010.

[4] G. Zhu, X. Lin, K. Zhu, W. Zhang, and J. X. Yu, "TreeSpan: Efficiently computing similarity all-matching," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, 2012, pp. 529–540.

[5] Y. Tian, R. C. McEachin, C. Santos, D. J. States, and J. M. Patel, "SAGA: A subgraph matching tool for biological graphs," *Bioinformatics*, vol. 23, no. 2, pp. 232–239, Jan. 2007.

[6] A. Sanfeliu and K.-S. Fu, "A distance measure between attributed relational graphs for pattern recognition," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-13, no. 3, pp. 353–362, May 1983.

[7] M. Stauffer, T. Tschachtli, A. Fischer, and K. Riesen, "A survey on applications of bipartite graph edit distance," in *Proc. GbRPR*, 2017, pp. 242–252.

[8] Z. Abu-Aisheh, R. Raveaux, J.-Y. Ramel, and P. Martineau, "A parallel graph edit distance algorithm," *Expert Syst. Appl.*, vol. 94, pp. 41–57, Mar. 2018.

[9] R. Wang, Y. Fang, and X. Feng, "Efficient parallel computing of graph edit distance," in *Proc. IEEE 35th Int. Conf. Data Eng. Workshops (ICDEW)*, Apr. 2019, pp. 233–240.

[10] Z. Zeng, A. K. H. Tung, J. Wang, J. Feng, and L. Zhou, "Comparing stars: On approximating graph edit distance," *Proc. VLDB Endowment*, vol. 2, no. 1, pp. 25–36, Aug. 2009.

[11] K. Riesen, S. Fankhauser, and H. Bunke, "Speeding up graph edit distance computation with a bipartite heuristic," in *Proc. MLG*, 2007, pp. 102–111.

[12] X. Zhao, C. Xiao, X. Lin, W. Wang, and Y. Ishikawa, "Efficient processing of graph similarity queries with edit distance constraints," *VLDB J.*, vol. 22, no. 6, pp. 727–752, Dec. 2013.

[13] J. Kim, D.-H. Choi, and C. Li, "Inves: Incremental partitioning-based verification for graph similarity search," in *Proc. EDBT*, 2019, pp. 229–240.

[14] L. Chang, X. Feng, X. Lin, L. Qin, W. Zhang, and D. Ouyang, "Speeding up GED verification for graph similarity search," in *Proc. IEEE 36th Int. Conf. Data Eng. (ICDE)*, Apr. 2020, pp. 793–804.

[15] Z. Abu-Aisheh, R. Raveaux, J.-Y. Ramel, and P. Martineau, "An exact graph edit distance algorithm for solving pattern recognition problems," in *Proc. Int. Conf. Pattern Recognit. Appl. Methods*, 2015, pp. 271–278.

[16] K. Gouda and M. Hassaan, "CSI_{GED}: An efficient approach for graph edit similarity computation," in *Proc. IEEE 32nd Int. Conf. Data Eng. (ICDE)*, May 2016, pp. 265–276.

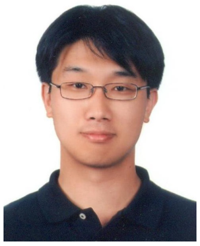
[17] X. Zhao, C. Xiao, X. Lin, and W. Wang, "Efficient graph similarity joins with edit distance constraints," in *Proc. IEEE 28th Int. Conf. Data Eng. ICDE*, Apr. 2012, pp. 834–845.

[18] J. Lerouge, Z. Abu-Aisheh, R. Raveaux, P. Héroux, and S. Adam, "Exact graph edit distance computation using a binary linear program," in *Proc. S SSPR*, 2016, pp. 485–495.

[19] K. Gouda and M. Hassaan, "A novel edge-centric approach for graph edit similarity computation," *Inf. Syst.*, vol. 80, pp. 91–106, Feb. 2019.

[20] X. Zhao, C. Xiao, X. Lin, Q. Liu, and W. Zhang, "A partition-based approach to structure similarity search," *Proc. VLDB Endowment*, vol. 7, no. 3, pp. 169–180, Nov. 2013.

[21] X. Zhao, C. Xiao, X. Lin, W. Zhang, and Y. Wang, "A partition-based approach to structure similarity search," *VLDB J.*, vol. 27, no. 1, pp. 55–78, 2018.



JONGIK KIM received the B.S. and M.S. degrees in computer science from the Korea Advanced Institute of Science and Technology (KAIST), in 1998 and 2000, respectively, and the Ph.D. degree in computer engineering from Seoul National University, South Korea, in 2004. From 2004 to 2007, he worked as a Senior Researcher with the Electronics and Telecommunications Research Institute (ETRI). He joined the Division of Computer Science and Engineering, Jeonbuk National University, as a Faculty Member, in 2007. From 2012 to 2013, he was a Visiting Scholar with the University of California at Irvine (UCI). He is currently a Professor with Jeonbuk National University. His research interests include semi-structured database (XML database), telematics systems, flash-memory data management, event stream processing, and similarity query processing.