A Distributed Algorithm for Graph Edit Distance

Zeina Abu-Aisheh, Romain Raveaux, Jean-Yves Ramel and Patrick Martineau

Laboratoire d'Informatique (LI), Université François Rabelais
37200, Tours, France
Email: f_author,s_author@univ-tours.fr

***Abstract*—Graph edit distance is an error-tolerant matching paradigm that can be used efficiently to address different tasks in pattern recognition, machine learning, and data mining. The literature is rich of many fast heuristics with unbounded errors but few works are devoted to exact graph edit distance computation. Exact graph edit distance methods suffer from high time and memory consumption. In the meantime, heavy computation tasks have moved from desktop applications to servers in order to spread the computation load on many machines. This paradigm leads to re-design methods in terms of scalability and performance. In this paper, a distributed and optimized branch-and-bound algorithm for exact graph edit distance computation is proposed. The search tree is cleverly pruned thanks to a lower and upper bounds' strategy. In addition, tree branches are explored in a completely distributed manner to speed up the tree traversal. Meaningful performance evaluation metrics are presented. Experiments were conducted on two publicly available datasets. Results demonstrate that under time constraints the most precise solutions were obtained by our method against five methods from the literature.**

Keywords–Pattern Recognition; Graph Matching; Graph Edit Distance; Branch-and-Bound; Distribution; Hadoop; MPI.

I. INTRODUCTION

Graph is an efficient data structure for object representation in structural pattern recognition (PR). Graphs can be divided into two main categories. First, graphs that are only based on their topological structures. Second, graphs with attributes on edges, vertices or both of them. Such attributes efficiently describe objects in terms of shape, color, coordinate, size, etc. and their relations [1]. The latter type of graphs is referred to as attributed graphs.

Representing objects by graphs turns the problem of object comparison into a graph matching (GM) one where evaluation of topological and/or statistical similarity of two graphs has to be found [2]. Researchers often shed light on error-tolerant GM, where an error model can be easily integrated into the GM process. The complexity of error-tolerant GM is NP-complete [3]. Consequently, the Graduated Assignment algorithm [4] has been employed to solve suboptimally error-tolerant GM. However, its complexity is $o(n^6)$ where n is number of vertices of both graphs. Several methods have a reduced complexity. However, they are not flexible as they do not cope with all the types of vertices and edges. For instance, the spectral methods [5], [6] deal with unlabeled graphs or only allow severely constrained label alphabets. Other methods are restricted to specific types of graphs [7]–[9], to name just a few.

Among error-tolerant problems for matching arbitrarily structured and arbitrarily attributed graphs, Graph Edit Distance (GED) is of great interest. GED is a discrete optimization problem. The search space of possible matching is represented as an ordered tree where a tree node is partial matching between two graphs. GED is NP-complete where its complexity is exponential in the number of vertices of the involved graphs [10]. GED can be applied to any type of graphs, including hypergraphs [11]. Many fast heuristic methods with unbounded errors have been proposed in the literature (e.g., [10]–[15]). On the other hand, few exact approaches have been proposed [16]–[18].

Recently, an exact Depth-First GED algorithm, referred to as *DF*, has been proposed in [18]. *DF* outperforms a well-known Best-First algorithm [16], referred to as A^*, in terms of memory consumption and run time. *DF* works well on relatively small graphs. To solve bigger matching problems, in this paper, we propose to extend *DF* to a distributed version which aims at spreading the workload over a cluster of machines. The distribution scheme is based on tree-decomposition and notification techniques. Instead of simply proposing a distributed *DF* algorithm using Message Passing Interface [19], we propose a master-slave distributed *DF* on top of Hadoop [20] with one synchronized variable achieved via ZooKeeper [21]. Roughly speaking, our algorithm consists of three main steps: First, a decomposition step where the master process divides the big matching problem into sub-problems. Second, a distribution stage where the master dispatches the sub problems among slave processes or so-called workers (i.e., processes to which a portion of work is associated). Third, a search-tree exploration step where each worker starts to explore its assigned problem by performing a partial *DF*. A notification step occurs when a worker succeeds in finding a better solution. In this case, the master informs the other workers and all of them update their upper bound. When a worker finishes the exploration of its assigned sub-problem, it asks the master for another one. Finally, the execution of the algorithm finishes when all the sub-problems generated in the decomposition stage are explored. The proposed algorithm is supported by novel evaluation performance metrics [22]. These metrics aim at comparing our algorithm with a set of GED approaches based on several significant criteria.

The rest of the paper is organized as follows. In Section II, the notations used in the paper are introduced. Moreover, the state of the art of GED approaches and distributed branch-and-bound techniques is presented. In Section III, the proposed distributed model is demonstrated. In Section IV, the datasets and the experimental protocols used to point out the performance of the proposed approaches are determined. Section V presents the obtained results and raises a discussion afterwards. Finally, conclusions are drawn and future perspectives are discussed in Section VI.

 ISBN: 978-1-61208-486-2

II. Related Works

In this section, we first define our basic notations and introduce GED and its computation.

A. Notations

1) Graph Based Notations: An attributed Graph (AG) is represented by a four-tuple, AG = (V, E, μ, ζ), where V and E are sets of vertices and edges such as $E \subseteq V \times V$. Both $\mu : V \rightarrow L_V$ and $\zeta : E \rightarrow L_E$ are vertex and edge labeling functions which associate an attribute or a label to each vertex v_i and edge e_i. L_V and L_E are unconstrained vertex and edge attributes sets, respectively. L_V and L_E can be given by a set of floats $L = \{1, 2, 3\}$, a vector space $L = \mathbb{R}^N$ and/or a finite set of symbolic attributes $L = \{x, y, b\}$.

GED is a graph matching method whose concept was first reported in [3], [23]. Its basic idea is to find the best set of transformations that can transform graph g_1 into graph g_2 by means of edit operations on graph g_2.

Let $g_1 = (V_1, E_1, \mu_1, \zeta_1)$ and $g_2 = (V_2, E_2, \mu_2, \zeta_2)$ be two graphs, GED between g_1 and g_2 is defined by:

$$GED(g_1, g_2) = \min_{ed_1, \cdots, ed_k \in \gamma(g_1, g_2)} \sum_{i=1}^{k} c(ed_i) \qquad (1)$$

where c denotes the cost function measuring the strength $c(ed_i)$ of an edit operation ed_i and $\gamma(g_1, g_2)$ denotes the set of edit paths transforming g_1 into g_2. The penalty, or cost, is dependent on the strength of the difference between the actual attributes information. Structure violations are also subject to a cost which is usually dependent on the magnitude of the structure violation [24]. And so, the penalty costs of each of deletion, insertion and substitution affect the matching process.

A standard set of edit operations is given by insertions, deletions and substitutions of both vertices and edges. We denote the substitution of two vertices u and v by $(u \rightarrow v)$, the deletion of vertex u by $(u \rightarrow \epsilon)$ and the insertion of vertex v by $(\epsilon \rightarrow v)$. For edges ($e.g.$ e and z), we use the same notations used for vertices. A complete edit path (EP) refers to an edit path that fully transforms g_1 into g_2 (i.e., complete solution). Mathematically, $EP = \{ed_i\}_{i=1}^{k}$.

2) Distribution Based Notations:

Definition II.1. Master-Slave Architecture
Master-Slave refers to an architecture in which one device (the master) controls one or more other devices (the slaves).

Definition II.2. Job
A job is a distributed procedure which has one or more *workers* (i.e., processes that are assigned to tasks). Each worker takes a task or a bunch of tasks to be solved.

B. Graph Edit Distance Computation

The methods of the literature can be divided into two categories depending on whether they can ensure the optimal matching to be found or not.

1) Exact Graph Edit Distance Approaches: A widely used method for edit distance computation is based on the A^* algorithm [25]. This algorithm is considered as a foundation work for solving GED. A^* is a Best-First algorithm where the enumeration of all possible solutions is achieved by means of an ordered tree that is constructed dynamically at run time by iteratively creating successor nodes. At each time, the node or so called partial edit path p that has the least $g(p) + h(p)$ is chosen. $g(p)$ represents the cost of the partial edit path accumulated so far while $h(p)$ denotes the estimated cost from p to a leaf node representing a complete edit path. The sum $g(p) + h(p)$ is referred to as a lower bound $lb(p)$. Given that the estimation of the future costs $h(p)$ is lower than, or equal to, the real costs, an optimal path from the root node to a leaf node is guaranteed to be found [26]. Leaf nodes correspond to feasible solutions and so complete edit paths. In the worst case, the space complexity can be expressed as $O(|\gamma|)$ [27] where $|\gamma|$ is the cardinality of the set of all possible edit paths. Since $|\gamma|$ is exponentional in the number of vertices involved in the graphs, the memory usage is still an issue.

To overcome the memory consumption problem of A^*, a recent Depth-First GED algorithm, referred to as *DF*, has been proposed in [18]. This algorithm speeds up the computations of GED thanks to its upper and lower bounds pruning strategy and its some associated preprocessing steps. Moreover, *DF* does not exhaust memory as the number of pending edit paths that are stored in the set, called *OPEN*, is relatively small thanks to the Depth-First search where the number of pending nodes is $|V_1|.|V_2|$ in the worst case.

2) Approximate Graph Edit Distance Approaches: Variants of approximate GED algorithms are proposed to make GED computation substantially faster. A modification of A^*, called Beam-Search (*BS*), has been proposed in [28]. Instead of exploring all edit paths in the search tree, only x most promising partial edit paths are kept in the set of promising candidates.

In [26], the problem of graph matching is reduced to finding an optimal matching in a complete bipartite GM, this algorithm is referred to as *BP*. In the worst case, the maximum number of operations needed by *BP* is $O(n^3)$. Since *BP* considers local structures rather than global ones, an overestimation of the exact GED cannot be neglected. A recent algorithm [29], named *FBP*, reduced the size of *BP*'s matrices. Recently, researchers have observed that *BP*'s overestimation is very often due to a few incorrectly assigned vertices. That is, only few vertex substitutions from the next step are responsible for additional (unnecessary) edge operations in the step after and thus resulting in the overestimation of the exact edit distance. Thus, recent works have been proposed to swap the misleading mappings [30], [31]. These improvements increase run times. However, they improve the accuracy of *BP*.

C. Distributed Branch-and-Bound Algorithms

Our interest in this paper is to propose a distributed extension of *DF* to be able to match large graphs. The best computing design that suits *DF* is SPMD [32] where a portion of the data (i.e., sub-search tree) is given to each process and all processes execute the proposed method on their associated sub-search trees. Since *DF* is a Branch-and-Bound (BnB) algorithm, we survey the state of art of distributed BnB approaches. However, before surveying the literature, some challenging questions should be listed:

- What is(are) the sub-task(s) associated to each process?
- What is the estimated time/needed memory per sub-task?

 ISBN: 978-1-61208-486-2

- What is the number of needed processes?
- How many sub-tasks one may generate before the distribution starts?
- How to efficiently distribute the search tree nodes of the irregular search tree among a large set of processes? Note that the decomposition, or distribution, can be irregular due to the bounding, or pruning with the help of *g(p)+h(p)*. Such a thing cannot be known except at run time.

These raised questions will be answered after exploring the state-of-art's methods dedicated to distributed BnB.

In [33], a one-iteration MPI approach was proposed. This approach is dedicated to solving three-phase electrical distribution networks. In the beginning, a specific number of nodes are generated by the master process. When this number is reached, no more nodes could be generated. The master then gives, or sends, a node to each slave. Then, each slave starts the exploration of the search tree in a Depth-First way. Once a slave finds a better upper bound, it sends to the master that updates all slaves. Once a slave finishes its the exploration of its assigned node, it sends a message to the master asking for a new node and the process continues. The drawback of such an approach is that once all the nodes, generated by the master, are given to all processes, some processes might become idle because they finished their associated nodes. Such a fact does not allow this approach to use all its resources at each time. In [34], a MPI BnB approach was proposed to solve the knapsack problem. This work is similar to [33]. However, the way of exploring the tree is left to the user so one can choose either Depth-First, a Best-First or a Breadth-First.

To the best of our knowledge, in the literature there is no distributed BnB method dedicated to solving GED. Both [33] and [34] are based on MPI which has no fault-tolerance. That is, if one slave process fails, one needs to re-execute all the processes. In this paper, instead of proposing an approach bases on MPI, we build MPI upon Hadoop [20]. Hadoop is tolerant to faults, thus, if one process fails, a master program selects another a free process to do its task. Roughly speaking, only the upper bound *UB* has to be shared with all processes. However, Hadoop is a model with restricted communication patterns. In order to allow processes to send messages and notify the other processes when finding a better *UB*, a message passing tool is adopted [21].

The search tree of GED contains nodes that represent partial edit paths. When thinking of a distributed approach of *DF*, these edit paths can be given to processes as tasks to be solved. Such a step divides the GED problem into smaller problems. The GED problem is irregular in the sense of having an irregular search tree where the number of nodes differs, depending on the ability of $lb(p)$ to prune the search tree. Based on that, it becomes hard to estimate the time needed by processes to explore a branch.

In [33] and [34], the authors did not mention the method they followed in order to generate nodes before the distribution starts. In our GED problem, one may think of A^* since it starts exploring nodes that lead to the optimal solution, if $lb(p)$ is carefully chosen. But, there are two key issues: First, how many nodes shall be generated by A^* before a *DF* procedure starts? Second, how to divide the nodes between processes?.

Input: Non-empty attributed graphs $g_1 = (V_1, E_1, \mu_1, v_1)$ and $g_2 = (V_2, E_2, \mu_2, v_2)$ where $V_1 = \{u_1, ..., u_{|v_1|}\}$ and $V_2 = \{v_1, ..., v_{|v_2|}\}$. A parameter N which is the number of the first generated partial edit paths.
Output: A minimum distance $UBCOST_{shared}$ and a minimum cost edit path (*UB*) from g_1 to g_2 e.g., *UB*= $\{u_1 \rightarrow v_3, u_2 \rightarrow \epsilon, \epsilon \rightarrow v_2\}$

1: (*UB* , *UBCOST*) $\leftarrow$ BP(g_1, g_2)
2: $OPEN \leftarrow \{\phi\}$
3: $\mathcal{Q} \leftarrow A^*(N)$
4: $\mathcal{Q} \leftarrow$ SortAscending($\mathcal{Q}$)
5: **for** $q \in \mathcal{Q}$ **do**
6: $OPEN$.AddFirst(q);
7: **end for**
8: $UBCOST_{shared} \leftarrow$ *UBCOST* {put *UBCOST* in an accessible place to all workers}
9: $UB_{shared} \leftarrow$ *UB* {put *UB* in an accessible place to all workers}
10: $File_{OPENshared} \leftarrow$ *OPEN*
11: **parallel for** $w \in W$ **do**
12: Get-Next-Task: $p \leftarrow File_{OPENshared}$.popFirst()
13: Call PartialDF(p,W,$UBCOST_{shared}$, UB_{shared})
14: **if** $File_{OPENshared}$ is not empty **then**
15: Repeat Get-Next-Task
16: **end if**
17: **end parallel for**
18: Return (UB_{shared}, $UBCOST_{shared}$).

Figure 1. Distributed *DF* (*D-DF*).

III. Distributed Depth-First GED Approach

Our distributed approach, referred to as *D-DF*, consists of a single job. Figure 1 represents the three main steps of *D-DF*. First, the master matches g_1 and g_2 using *BP* and outputs both the matching sequence *UB* and its edit distance *UBCOST* (line 1). Second, A^* is executed and stopped once N partial edit paths are generated (line 3). Afterwards, these partial edit paths ($\mathcal{Q}$) are sorted in ascending order and inserted to *OPEN* (lines 4 to 7). The master also saves *UB* and its *UBCOST* in a place/space accessible by all workers W (lines 8 and 9). Finally, the master distributes the work (i.e., $\mathcal{Q}$) among workers, each worker takes one edit path from the master at a time (line 12). This step adapts the dynamic scheduling where tasks are associated to processes at run time. Thus, each process takes one and only one edit path at a time t instead of having a predefined list of edit paths. Workers start the exploration of their associated partial edit paths (line 13). If a worker finishes its assigned partial edit path, it sends a message to the master asking for a new edit path (lines 14 to 16). When finishing all the partial edit paths, saved in $File_{OPENshared}$, the program outputs UB_{shared} and $UBCOST_{shared}$ as an optimal solution of matching g_1 and g_2 (line 18).

Figure 2 demonstrates the function *PartialDF* that each worker w executes on its assigned partial edit path p. Note that each p is given to an available worker by the master. The procedures of this algorithm are similar to *DF*, the only difference is that *UBCOST* and *UB* are saved in a shared space that is accessible by all workers W. These shared variables are

Input: An edit path p, the set of workers W, $UBCOST_{shared}$ and UB_{shared}

```
1: OPEN ← {φ}, p_min ← φ
2: OPEN.addFirst(p)
3: r ← parent(u_1), r_tmp ← r
4: UBCOST ← read(UBCOST_shared)
5: Set watch on UBCOST_shared
6: while OPEN != {φ} do
7:     p ← OPEN.popFirst()        ▷ Take first element and remove it from OPEN
8:     Listp ← GenerateChildren(p)
9:     if Listp = {φ} then
10:        for v_i ∈ pendingV_2(p) do
11:            q ← insertion(ϵ , v_i)        ▷ i.e., {ϵ → v_i}
12:            p.AddFirst(q)
13:        end for
14:        if g(p) < UB then
15:            UB ← g(p), Bestedit path ← p
16:            UBCOST_shared ← g(p) + h(p)
17:            UB_shared ← p
18:            MASTER: notify-all-workers w ∈ W
19:            for w ∈ W do
20:                UBCOST ← read(UBCOST_shared)
21:                Reset watch on UBCOST_shared
22:            end for
23:        end if
24:    else
25:        Listp ← SortAscending(Listp)        ▷ according to g(p)+h(p)
26:        for q ∈ Listp do
27:            if g(q) + h(q) < UB then
28:                OPEN.AddFirst(q)
29:            end if
30:        end for
31:    end if
32: end while
```

Figure 2. Function PartialDF.

referred to as $UBCOST_{shared}$ and UB_{shared}. All the workers read the value stored in $UBCOST_{shared}$ through *read* message (line 4). They also put a watch on $UBCOST_{shared}$ via *Set-Watch* message so as to be awaken when any change happens to its value (line 5). All the workers solve their associated partial edit path. Whenever worker w succeeds in finding a better value of its *UBCOST*, it updates both $UBCOST_{shared}$ and UB_{shared} through *update* messages (lines 15 and 17), the master then sends a notification via *notify-Worker* message to all the other workers (line 18). Workers read the new value, update their local *UB* and continue solving their problems. Moreover, workers re-establish, or reset, the watch for data changes through *Reset-Watch* message (lines 19 to 22). The update of $UBCOST_{shared}$ is done carefully as only one worker can change $UBCOST_{shared}$ at any time t. That is, if two workers want to change $UBCOST_{shared}$ at the same time, one of them is delayed by the master for some milliseconds before entering the critical point. The final answers (i.e., the optimal matching and its distance) are found in UB_{shared} and $UBCOST_{shared}$ respectively when all workers finish their associated tasks.

A. Advantages and Drawbacks

D-DF is a fully distributed approach where each worker accomplishes its task without waiting for each other. Moreover, the search tree is cleverly pruned. As soon as any worker finds a better $UBCOST_{shared}$, it sends the new value to the master. Then, the notification to all the other workers is achieved by the master. Finally, all the workers receive the new value. Such operations help at pruning the workers' search trees as fast as possible. *D-DF* is a single-job approach and thus the drawback behind such an approach is that some workers might become idle because there is no more edit path in $File_{OPENshared}$ while the other ones are still working as they have not finished their assigned edit paths. To overcome such a problem, in future work, this algorithm can be transformed into a multi-jobs, or multi-iteration, algorithm.

IV. Experiments

A. Environment

D-DF is built on top of Hadoop [20] with a notification tool called *ZooKeeper* [21] used to share $UBCOST_{shared}$ and UB_{shared} with all workers. Synchronizing these variables does not break the scalability. On the contrary, it helps in pruning the search tree as fast as possible. The evaluation of *D-DF* is conducted on 5 machines running Hadoop MapReduce version 1.0.4. Each node contains a 4-core Intel i7 processor 3.07GHz, 8GB memory and one hard drive with 380GB capacity. Hadoop was allocated 20 workers (4 workers per machine), each with a maximum JVM memory size of 1GB. Hadoop Distributed File System is used for dispatching edit paths with a replication factor that is equal to 3. For sequential algorithms, evaluations are conducted on one machine.

B. Studied Methods

We compare *D-DF* with five GED algorithms from the literature. From the related work, we chose two exact methods and three approximate ones. On the exact methods' side, we have chosen, A* and *DF*. In both algorithms *h(p)* is calculated by applying *BP* on vertices and edges in a separated manner [18]. On the approximate methods' side, we include *BS-1*, *BS-10*, *BS-100*, *BP* and *FBP*, see Section II-B.

C. Datasets

Recently, a new repository has been put forward to test the scalability of graphs [22]. Databases are divided into subsets each of which represents graphs with the same number of vertices. In this work, we use two PR datasets (GREC and Mutagenicity (MUTA)) taken from the repository. However, we eliminate the easy graph matching problems from both datasets since we are interested in difficult problems for distribution issues. To filter these databases, we run *DF* on each pair of graphs and stop it after 5 minutes. If there is no optimal solution found within 5 minutes, then the matching problem is considered as difficult. This results in 627 problems on MUTA and 92 problems on GREC. For more details about these datasets, please visit the GDR4GED repository [35].

D. Protocol

First, the effect the variable N (number of initial edit paths) is tested on several values. Five values of N are chosen: 20, 100, 250, 500 and 1000, where N=20 represents the least distributed case (i.e., one edit path per worker), N= 100 and

 ISBN: 978-1-61208-486-2

250 moderately amortize the communication between master and slaves whereas N=500 and 1000 is the more complete case where workers have to cross the network many times in order to ask for a new edit path once they finish solving an already assigned one. We also study the effect of increasing the number of machines, from 2 to 5 machines, on run time. Both tests are evaluated on GREC-20 (i.e., graphs of GREC whose number of vertices is twenty) [22].

The deviation was chosen as a metric to compare all the included methods [22]. We compute the error committed by each method m over the reference distances. For each pair of graphs matched by method m, we provide the following deviation measure:

$$dev(g_i, g_j)^m = \frac{|d(g_i, g_j)^m - R_{g_i,g_j}|}{R_{g_i,g_j}}, \ \forall (i,j) \in [\![1, G]\!]^2, \ \forall m \in \mathcal{M} \quad (2)$$

where G is the number of graphs. $d(g_i, g_j)^m$ is the distance obtained when matching g_i and g_j using method m while R_{g_i,g_j} corresponds to the best known solution among all the included methods. We also measure the overall time in milliseconds (ms), for each GED computation, including all the inherits costs computations. The mean run time is calculated per subset s and for each method m. Due to the high complexity of GED methods, we propose to evaluate them under a time constraint C_T that is equal to 300 seconds and a memory constraint C_M that is equal to 1GB. Note that the only algorithm that violates memory is A^*.

V. Results and Discussions

Figure 3 depicts the parameters study. One can see that increasing the number of machines decreases the run time. For some instances, the runtime was not reduced due to the difficulty of matching problems. As for N, the best case was when it equals 250, owing to moderately performing tasks-dispatch. These values were used for the rest of experiments.

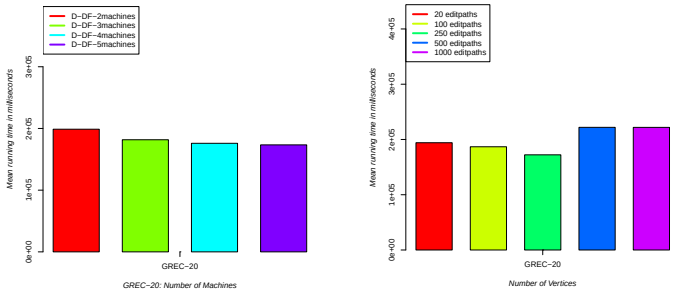

Figure 3. Parameters: Left (Number of Machines), Right (Variable N). The objective of this study is to choose the values that minimize *D-DF*'s run time.

Figure 4 illustrates the deviation of the included methods on GREC and MUTA. Figure 4(a) shows that *D-DF* has the least deviation (0%) on all subsets, followed by *DF*. However, that was not the case on MUTA, see Figure 4(b). *BS100* outperformed *D-DF* in terms of number of best found solutions. The major differences between these algorithms are a) The search space exploration manner and b) the Vertices-Sorting strategy which is adapted in *DF* [18] and not in BS. In fact, *BP* is integrated in the preprocessing step of *DF* to sort vertices of g_1. Since *BP* did not give a good estimation on MUTA, it was also irrelevant when sorting the vertices of g_1 resulting in the exploration of misleading nodes in the search tree. Since the graphs of MUTA are relatively large, backtracking nodes took time. MUTA contains symbolic attributes while *DF* and *D-DF* are designed for rich attributed graphs where the use of *BP* in the vertices sort is meaningful. However, the deviation of *BS* and *D-DF* was relatively similar. Both *D-DF* and *DF* succeeded in finding upper bounds that are better than *BP*. *D-DF*, however, has always outperformed *DF* in terms of deviation. This can be remarkably seen on MUTA where, in average, the deviation of *DF* was 18% while the deviation of *D-DF* was 6.5%.

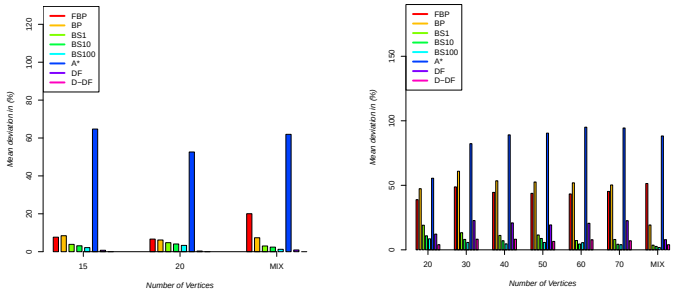

Figure 4. Deviation: Left (GREC), Right (MUTA). Note that the lower the deviation the better the algorithm.

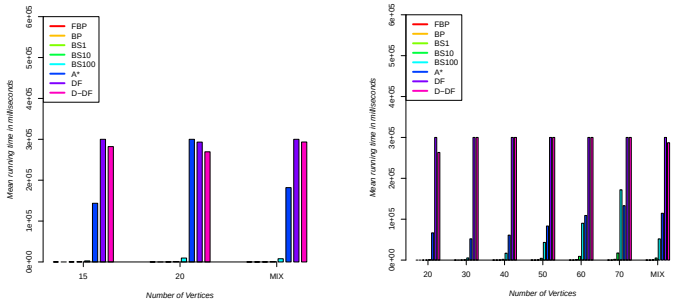

Figure 5. Run time: Left (GREC), Right (MUTA). Note that the lower the run time the better the algorithm.

D-DF was always faster or equal to *DF*, see Figure 5. At a first glance, one can think that A^* was faster than both *DF* and *D-DF*. However, that was not the case. In fact, A^* was unable to output feasible solutions and was stopped because of its memory bottleneck. *BP* was the fastest algorithm (12.3 milliseconds on GREC and 11 milliseconds on MUTA).

VI. Conclusion and Perspectives

In the present paper, we have considered the problem of GED computation for PR. GED is a powerful and flexible paradigm that has been used in different applications in PR. In the literature, few exact GED algorithms have been proposed. Recently, a Depth-First GED algorithm (*DF*) has shown to be effective. *DF*, thanks to its Depth-First exploration, upper and lower bounds pruning strategies, overcomes the high memory consumption from which a well-known A^* algorithm suffers. However, *DF* can match relatively small graphs. In this paper, we have proposed to extend it to a distributed version called *D-DF*. We build a master-slave architecture over Hadoop in order to take advantage of the fault-tolerance of Hadoop. Each

 ISBN: 978-1-61208-486-2

worker gets one partial edit path and all workers solve their assigned edit paths in a fully distributed manner. In addition, a notification process is integrated. When any worker finds a better upper bound, it notifies the master to share the new upper bound with all workers.

In the experiments part, we have proposed to evaluate both exact and approximate GED approaches, using novel performance evaluation metrics under time and memory constraints, on two different datasets (GREC and MUTA). Experiments have pointed out that *D-DF* has the minimum deviation. *BS* is slightly superior to *D-DF* in terms of deviation on the MUTA dataset. In fact, one weakness of *DF* and so *D-DF* is that their sorting strategy is *BP*-dependent. One solution could be to better learn the upper bound and so the sorting strategy in function of dataset type/nature. Experiments have also demonstrated that *D-DF* always outperforms *DF* in terms of deviation and run time. Indeed, *D-DF* is flexible as one can add more machines and thus decrease the running time. Results have also indicated that there is always a trade-off between deviation and running time. In other words, approximate methods are fast, however, they are not as accurate as exact ones. On the other hand, *DF* and *D-DF* take longer time but lead to better results.

The main drawback behind *D-DF* is that it is a single-job algorithm. When there is no time constraint, some workers work while others may become idle after finishing the exploration of their assigned partial edit paths. To overcome this drawback and as future work, we aim at transforming *D-DF* into a multi-iteration method where all workers work without becoming idle. Moreover, two ideas can be applied for both *DF* and *D-DF*. First, coming up with a better lower bound and thus making the calculations faster. Second, learning to sort the vertices of each dataset in a way that minimizes its deviation. Such an extension of *DF* and *D-DF* can beat the approximate approaches when matching graphs of MUTA.

REFERENCES

[1] K. Riesen and H. Bunke, "Iam graph database repository for graph based pattern recognition and machine learning," 2008, pp. 287–297.

[2] M. Vento, "A long trip in the charming world of graphs for pattern recognition," Pattern Recognition, vol. 48, no. 2, 2015, pp. 291–301.

[3] H. Bunke, "Inexact graph matching for structural pattern recognition," Pattern Recognition Letters, vol. 1, no. 4, 1983, pp. 245–253.

[4] S. Gold and A. Rangarajan, "A graduated assignment algorithm for graph matching," Pattern Analysis and Machine Intelligence, IEEE Transactions on, vol. 18, 1996, pp. 377–388.

[5] S. Umeyama, "An eigendecomposition approach to weighted graph matching problems," Pattern Analysis and Machine Intelligence, IEEE Transactions on, vol. 10, 1988, pp. 695–703.

[6] R. Wilson, E. Hancock, and B. Luo, "Pattern vectors from algebraic graph theory," Pattern Analysis and Machine Intelligence, IEEE Transactions on, vol. 27, 2005, pp. 1112–1124.

[7] X. Jiang and H. Bunke, "Optimal quadratic-time isomorphism of ordered graphs," Pattern Recognition, vol. 32, no. 7, 1999, pp. 1273 – 1283.

[8] J. E. Hopcroft and J. K. Wong, "Linear time algorithm for isomorphism of planar graphs (preliminary report)," in Proceedings of the Sixth Annual ACM Symposium on Theory of Computing. New York, NY, USA: ACM, 1974, pp. 172–184.

[9] M. Neuhaus and H. Bunke, "An error-tolerant approximate matching algorithm for attributed planar graphs and its application to fingerprint classification," in SSPR WORKSHOP. LNCS 3138. Springer, 2004, pp. 180–189.

[10] Z. Zeng, A. K. H. Tung, J. Wang, J. Feng, and L. Zhou, "Comparing stars: On approximating graph edit distance," vol. 2, 2009, pp. 25–36.

[11] W.-H. Tsai and K.-S. Fu, "Error-correcting isomorphisms of attributed relational graphs for pattern analysis," Systems, Man and Cybernetics, IEEE Transactions on, vol. 9, no. 12, 1979, pp. 757–768.

[12] J. K. W. Christmas and M. Petrou., "Structural matching in computer vision using probabilistic relaxation." IEEE Trans. PAMI,, vol. 2, 1995, pp. 749–764.

[13] A. D. J. Cross and E. R. Hancock, "Graph matching with a dual-step em algorithm," IEEE Trans. Pattern Anal. Mach. Intell., vol. 20, 1998, pp. 1236–1253.

[14] e. a. Finch, Wilson, "An energy function and continuous edit process for graph matching," Neural Computat, vol. 10, 1998, pp. 1873–1894.

[15] P. Kuner and B. Ueberreiter, "Pattern recognition by graph matching: Combinatorial versus continuous optimization," International journal in Pattern Recognition and Artificial Intelligence, vol. 2, 1988, pp. 527—-542.

[16] P. Hart, N. Nilsson, and B. Raphael., "A formal basis for the heuristic determination of minimum cost paths." IEEE Transactions of Systems, Science, and Cybernetics., vol. 28, 2004, pp. 100–107.

[17] D. Justice and A. Hero, "A binary linear programming formulation of the graph edit distance," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 28, 2006, pp. 1200–1214.

[18] Z. Abu-Aisheh, R. Raveaux, J.-Y. Ramel, and P. Martineau, "An exact graph edit distance algorithm for solving pattern recognition problems," Proceedings of ICPRAM, 2015, pp. 271–278.

[19] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, MPI-The Complete Reference, volume 1: The MPI Core, 2nd ed. Cambridge, MA, USA: MIT Press, 1998.

[20] T. White and D. Cutting, Hadoop : the definitive guide. O'Reilly, 2009.

[21] F. Junqueira and B. Reed, Zookeeper: Distributed Process Coordination, 2013.

[22] Z. Abu-Aisheh, R. Raveaux, and J.-Y. Ramel, "A graph database repository and performance evaluation metrics for graph edit distance," in Graph-Based Representations in Pattern Recognition - GbRPR 2015., 2015, pp. 138–147.

[23] A. Sanfeliu and K. Fu, "A distance measure between attributed relational graphs for pattern recognition," IEEE Transactions on Systems, Man, and Cybernetics, vol. 13, 1983, pp. 353–362.

[24] K. Riesen and H. Bunke, Graph Classification and Clustering Based on Vector Space Embedding. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 2010.

[25] S. Fankhauser, K. Riesen, and H. Bunke, "Speeding up graph edit distance computation with a bipartite heuristic," no. 6658, 2011, pp. 102–111.

[26] B. H. Riesen, K., "Approximate graph edit distance computation by means of bipartite graph matching." Image and Vision Computing., vol. 28, 2009, pp. 950–959.

[27] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, Third Edition, 3rd ed. The MIT Press, 2009.

[28] H. B. M. Neuhaus, K. Riesen, "Fast suboptimal algorithms for the computation of graph edit distance," Proceedings of SSPR., 2006, pp. 163–172.

[29] F. Serratosa, "Computation of graph edit distance: Reasoning about optimality and speed-up," Image and Vision Computing, vol. 40, 2015, pp. 38–48.

[30] K. Riesen and H. Bunke, "Improving Approximate Graph Edit Distance by Means of a Greedy Swap Strategy," vol. 8509, 2014, pp. 314–321.

[31] K. Riesen, A. Fischer, and H. Bunke, "Improving approximate graph edit distance using genetic algorithms," 2014, pp. 63–72.

[32] M. J. Atallah and S. Fox, Eds., Algorithms and Theory of Computation Handbook, 1st ed. Boca Raton, FL, USA: CRC Press, Inc., 1998.

[33] L. Barreto and M. Bauer, "Parallel branch and bound algorithm - a comparison between serial, openmp and mpi implementations," journal of Physics: Conference Series, vol. 256, no. 5, 2010, pp. 012–018.

[34] I. Dorta, C. León, and C. Rodríguez, "A comparison between mpi and openmp branch-and-bound skeletons." in IPDPS, 2003.

[35] "Gdr4ged," http://www.rfai.li.univ-tours.fr/PublicData/GDR4GED/home.html., 2015.

 ISBN: 978-1-61208-486-2