A parallel graph edit distance algorithm

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

A parallel graph edit distance algorithm

Zeina Abu-Aisheh[a], Romain Raveaux[a], Jean-Yves Ramel[a], Patrick Martineau[a]

[a]*Laboratoire d'Informatique (LI), Université François Rabelais, 37200, Tours, France*
Email addresses: firstname.lastname@univ-tours.fr

Abstract

Graph edit distance (GED) has emerged as a powerful and flexible graph matching paradigm that can be used to address different tasks in pattern recognition, machine learning, and data mining. GED is an error-tolerant graph matching problem which consists in minimizing the cost of the sequence that transforms a graph into another by means of edit operations. Edit operations are deletion, insertion and substitution of vertices and edges. Each vertex/edge operation has its associated cost defined in the vertex/edge cost function. Unfortunately, Unfortunately, the GED problem is NP-hard. The question of elaborating fast and precise algorithms is of first interest. In this paper, a parallel algorithm for exact GED computation is proposed. Our proposal is based on a branch-and-bound algorithm coupled with a load balancing strategy. Parallel threads run a branch-and-bound algorithm to

[☆]Fully documented templates are available in the elsarticle package on CTAN.

Email addresses: support@elsevier.com (Romain Raveaux), support@elsevier.com (Jean-Yves Ramel), support@elsevier.com (Patrick Martineau)

URL: www.elsevier.com (Zeina Abu-Aisheh)

explore the solution space and to discard misleading partial solutions. In the mean time, the load balancing scheme ensures that no thread remains idle. Experiments on 4 publicly available datasets empirically demonstrated that under time constraints our proposal can drastically improve a sequential approach and a naive parallel approach. Our proposal was compared to 6 other methods and provided more precise solutions while requiring a low memory usage.

1. Introduction

Attributed graphs are powerful data structures for the representation of structured entities. In a graph-based representation, vertices and their attributes describe objects (or part of objects) while edges represent interrelationships between the objects. Due to the inherent genericity of graph-based representations, and thanks to the improvement of computer capacities, structural representations have become more and more popular in the field of Pattern Recognition.

Graph edit distance (GED) is a graph matching paradigm whose concept was first reported in (Sanfeliu and Fu, 1983). The basic idea is to find the best sequence of edit operation to transform a graph G_1 into another graph G_2. The allowed operations are insertion, deletion and/or substitution of vertices and their corresponding edges. GED can be used as a dissimilarity measure for arbitrarily structured and arbitrarily attributed graphs. In contrast to other approaches, it does not suffer from any restrictions and can be applied

to any type of graph (including hypergraphs (H. Bunke, 1983)). The main drawback of GED is its computational complexity which is exponential in the number of vertices of the involved graphs.

Many fast heuristic GED methods have been proposed in the literature (W. Christmas and Petrou., 1995; Zeng et al., 2009; Fankhauser et al., 2012; Fischer et al., 2013; Serratosa, 2015; Ferrer et al., 2015; Bougleux et al., 2016). However, these heuristic algorithms can only find unbounded suboptimal values. On the other hand, only few exact approaches have been proposed (Tsai et al., 1979; Justice D, 2006; Riesen et al., 2007; Abu-Aisheh et al., 2015).

Parallel computing has been fruitfully employed to handle time-consuming operations. Research results in the area of parallel algorithms for solving machine learning and computer vision problems have been reported in (Kumar et al., 1990). These researches demonstrated that parallelism can be exploited efficiently in various machine intelligence and vision problems such as deep learning (Deng and Yu, 2014) or fast fourier transform (Van Loan, 1992). In this paper, we take benefit of parallel computing to solve the exact GED problem.

The main contribution of this paper is an exact parallel algorithm based on a load balancing strategy for solving the GED problem. This paper lies in the idea that a parallel execution can help to converge faster to the optimal solution. Our method is very generic and can be applied to directed or undirected fully attributed graphs (i.e., with attributes on both vertices and

edges). By limiting the run-time, our exact method provides (sub)optimal solutions and becomes an efficient upper bound approximation of GED. A complete comparative study is provided where 6 exact and approximate GED algorithms were compared on 4 graph datasets. By considering both the quality of the proposed solutions and the speed of the algorithm, we show that our proposal is a good choice when a fast decision is required as in a classification context or when the time is less a matter but a precised solution is required as in image registration.

This paper is organized as follows: Section 2 presents the important definitions necessary for introducing our GED algorithm. Then, Section 3 reviews the existing approximate and exact approaches for computing GED. Section 4 describes the proposed parallel scheme based on a load balancing paradigm. Section 5 presents the experiments and analyses the obtained results. Section 6 provides some concluding remarks.

2. Problem Statements

In this section, we first introduce the GED problem which is formally defined as an optimization problem. Secondly, to cope with the inherent complexity of the GED problem, the use of parallel computing is argued. However, the parallel execution of a combinatorial optimization problem is not trivial and consequently the load balancing question is being raised. Finally, the load balancing problem is formally defined and presented to establish the basement of an efficient parallel algorithm.

2.1. Graph Edit Distance Problem

Attributed graph is defined as a tuple of 4 sets (V, E, μ, ζ) such that:

Definition 1. *Attributed Graph*
$G = (V, E, \mu, \zeta)$
V *is a set of vertices*
E *is a set of edges such as* $E \subseteq V \times V$
$\mu : V \rightarrow L_V$ *such that* μ *is a vertex labeling function which associates a label* l_V *to a vertex* v_i *with* $v_i \in V$, $\forall i \in [1,|V|]$
$\zeta : E \rightarrow L_E$ *such that* ζ *is an edge labeling function which associates a label* l_E *to an edge* e_i *with* $e_i \in E$, $\forall i \in [1,|E|]$

Definition 1 allows to handle arbitrarily structured graphs with unconstrained labeling functions. Labels for both vertices and edges can be given by a set of integers $\mathrm{L} = \{1, 2, 3, \cdots\}$, a vector space $\mathrm{L} = \mathbb{R}^n$ and/or a finite set of symbolic labels $\mathrm{L} = \{x, y, z, \cdots\}$.

GED is an error-tolerant graph matching paradigm, it defines the dissimilarity of two graphs by the minimum amount of distortion needed to transform one graph into another (H. Bunke, 1983). GED requires that each vertex/edge of graph G_1 is mapped to a distinct vertex/edge of graph G_2 or to a dummy vertex/edge. This dummy elements can absorb structural modifications between the involved two graphs. More formally, GED can be defined as follows:

Definition 2. *Graph Edit Distance Problem*

$$GED(G_1, G_2) = \min_{\gamma \in \Gamma(G_1, G_2)} \sum_{o \in \gamma} c(o)$$

Where $\Gamma(G_1, G_2)$ *denotes the set of edit paths transforming* $G_1 = (V_1, E_1, \mu_1, \zeta_1)$ *into* $G_2 = (V_2, E_2, \mu_2, \zeta_2)$*, and* c *denotes the cost function measuring the*

strength $c(o)$ of edit operation o.

A sequence of edit operations γ that transforms a graph G_1 into a graph G_2 is commonly referred to as edit path between G_1 and G_2. In order to represent the degree of modification imposed on a graph by an edit path, a cost function is introduced measuring the strength of the distortions caused by each edit operation. Consequently, the edit distance between graphs is defined by the minimum cost edit path between two graphs. Note that the edit operations on edges can be inferred by edit operations on their adjacent vertices, i.e., whether an edge is substituted, deleted, or inserted, depends on the edit operations performed on its adjacent vertices. An elementary edit operation o is one of vertex substitution ($v_1 \rightarrow v_2$), edge substitution ($e_1 \rightarrow e_2$), vertex deletion ($v_1 \rightarrow \epsilon$), edge deletion ($e1 \rightarrow \epsilon$), vertex insertion ($\epsilon \rightarrow v_2$) and edge insertion ($\epsilon \rightarrow e_2$) with $v_1 \in V_1$, $v_2 \in V_2$, $e_1 \in E_1$ and $e_2 \in E_2$. ϵ is a dummy vertex or edge which is used to model insertion or deletion. $c(.)$ is a cost function on elementary edit operations o. The cost function $c(.)$ is of first interest and can change the problem being solved. In (Bunke, 1997; Brun, 2012) a particular cost function for GED was introduced, and it was shown that under this cost function, GED computation is equivalent to the maximum common subgraph problem. Neuhaus and Bunke (Neuhaus and Bunke., 2007) showed that if each elementary operation satisfies the criteria of a distance (separability, symmetry and triangular inequality) then GED is metric. Recently, methods to learn the matching edit cost between graphs

have been published (Cortés and Serratosa, 2015). The discussion around the cost functions is beyond the topic of this paper that essentially focuses on the GED computation.

2.2. From GED Problem to Load Balancing Problem

GED is a discrete optimization problem that faces the combinatorial explosion curse. The complexity of GED was proven to be NP-hard where the computational complexity of matching is exponential in the number of vertices of the involved graphs (Zeng et al., 2009). At run time, the evolution of the size and shape of the search space is irregular and unpredictable. The search space is represented as an ordered tree.

For the sake of clarity in the rest of the paper, the term **vertex** refers to an element of a graph while the term **tree node** or **node** represents an element of the search tree.

The initial and leaf tree nodes correspond to the initial state and the final acceptable state in the search tree, respectively. Each edge represents a possible way of state change. A combinatorial optimization problem is essentially the problem of finding a minimum-cost path from an initial node to a leaf node in the search tree. More concretely in GED, each tree node is a sequence of edit operations. Leaf nodes are complete edit operations sequences (edit path) while intermediate nodes are partial solutions representing partial edit path. An example of search tree corresponding to GED computation is shown in Figure 1.

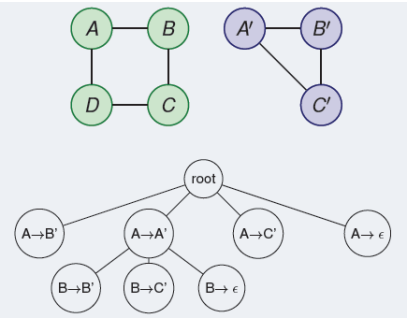


Figure 1: An incomplete search tree example for solving the GED problem. The first floor represents possible matchings of vertex A with each vertex of the second graph (in blue). A tree node is a partial solution which is to say a partial edit path.

The parallelism of combinatorial optimization problems is not trivial. In a parallel combinatorial search application, each thread searches for optimal solutions within a portion of the solution space. The shape and size of the search space change as the search proceeds. Consequently, tree nodes are generated and destroyed at run-time.

Portions that encompass the most promising solutions with high probability are expanded in priority and explored exhaustively, while portions that have unfruitful solutions are discarded at run-time. To ensure that parallel threads are always busy, tree nodes have to be dispatched at run-time. Hence, the local workload of a thread is difficult to predict.

The parallel execution of combinatorial optimization problems relies on load balancing strategies to divide the global workload of all threads iteratively at run-time. From the viewpoint of a workload distribution strategy, parallel optimizations fall in the *asynchronous* communication category where no thread waits another thread to finish in order to start a new task

(Bertsekas and Tsitsiklis, 1997). A thread initiates a balancing operation when it becomes lightly loaded or overloaded. The objective of the data distribution strategies is to ensure a fast convergence to the optimal solution such that all the tree nodes are evaluated as fast as possible. In this paper, we propose a parallel GED approach equipped with a load balancing strategy. This approach ensures that all threads have the same amount of load and all threads explore the most promising tree nodes first.

2.3. Load Balancing Problem

A parallel program is composed of multiple *threads*, each thread is a processing unit which processes one task. Multiple threads can exist within the same process and share resources such as memory (i.e., the values of its variables at any given moment). In our GED case, the task is to evaluate a tree node (a partial edit path) and to generate its children (the next possible states). A thread performs a task on a set of works. A *work* is a tree node characterized by its workload. A work cannot be shared between threads, it is the smallest unit of concurrency the parallel program can exploit.

Creating a parallel program involves first decomposing the overall computation into works and then assigning the works to threads. The decomposition together with the assignment steps is often called partitioning. The assignment on its own is referred to static load balancing.

Load balancing algorithms can be broadly categorized into two families: Static and dynamic.

2.3.1. Static load balancing

Static load balancing algorithms distribute works to threads once and for all, in most cases relying on a priori knowledge about the works and the system on which they run. These algorithms rely on the estimated execution times of works and inter-thread communication requirements. It is not satisfactory for parallel programs that dynamic and/or unpredictable. The problem of static load balancing is known to be NP-hard when the number of threads is greater or equal to 2 (Drozdowski, 2009).

2.3.2. Dynamic load balancing

Dynamic load balancing algorithms bind works to threads at run-time. Generally, a dynamic load balancing algorithm consists of three components: A load measurement rule, an initiation rule and a load balancing operation. A very detailed definition of load balancing models can be found in (Xu and Lau, 1997).

Load Measurement. Dynamic load balancing algorithms rely on the workload information of threads. The workload information is typically quantified by a load index, a non-negative variable which is equal to zero if the thread is idle or takes an increasing positive value when the load increases (Xu and Lau, 1997). Since the measure of workload would occur frequently, its calculation must be efficient.

Initiation Rule. This rule dictates when to initiate a load balancing operation. The execution of a balancing operation incurs non-negligible overhead;

its invocation must weight its overhead cost against its expected performance benefit. An initiation policy is thus needed to determine whether a balancing operation will be profitable.

Load Balancing Operation. This operation is defined by three rules: Location, distribution and selection rules. The location rule determines the partners of the balancing operation, i.e., the threads to involve in the balancing operation. The distribution rule determines how to redistribute workload among the selected threads. The selection rule selects the most suitable data for transfer among threads.

3. Related Work

In this section, an overview of the GED methods presented in the literature is given. Since our goal is to speed up the calculations of GED, parallelism is highly required. Therefore, we also cover the parallel methods, dedicated to solving branch-and-bound (BnB) problems, aiming at getting inspired by some of these works for parallelizing the GED calculations.

3.1. State-of-the-art of Graph Edit Distance

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.

3.1.1. Exact Graph Edit Distance Approaches

A widely used method for edit distance computation is based on the A^* algorithm (Riesen et al., 2007). 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 where $g(p)$ represents the cost of the partial edit path accumulated so far whereas $h(p)$ 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 (Riesen, 2009). Leaf nodes correspond to feasible solutions and so complete edit paths. In the worst case, the space complexity can be expressed as $O(|\Gamma|)$ (Cormen et al., 2009) where $|\Gamma|$ is the cardinality of the set of all possible edit paths. Since $|\Gamma|$ is exponential in the number of vertices involved in the graphs, the memory usage is still an issue.

To overcome the A^* problem, a recent depth-first BnB GED algorithm, referred to as *DF*, has been proposed in (Abu-Aisheh et al., 2015). This algorithm speeds up the computations of GED thanks to its upper and lower bounds pruning strategy and its preprocessing step. Moreover, this algorithm does not exhaust memory as the number of pending edit paths that are stored at any time t is relatively small thanks to the space complexity which is equal to $|V_1|.|V_2|$ in the worst case.

In both A^* and *DF*, $h(p)$ can be estimated by mapping the unprocessed

vertices and edges of graph G_1 to the unmapped to those of graph G_2 such that the resulting cost is minimal. The unprocessed edges of both graphs are handled separately from the unprocessed vertices. This mapping is done in a faster way than the exact computation and should return a good approximation of the true future cost. Note that the smaller the difference between $h(p)$ and the real future cost, the fewer nodes will be expanded by A^* and *DF*.

Almohamad and Duffuaa in (Almohamad and Duffuaa, 1993) proposed the first linear programming formulation of the weighted graph matching problem. It consists in determining the permutation matrix minimizing the L_1 norm of the difference between adjacency matrix of the input graph and the permuted adjacency matrix of the target one. More recently, Justice and Hero (Justice D, 2006) also proposed a binary linear programming formulation of the graph edit distance problem. GM is treated as finding a subgraph of a larger graph known as the edit grid. The edit grid only needs to have as many vertices as the sum of the total number of vertices in the graphs being compared. One drawback of this method is that it does not take into account attributes on edges which limits the range of application.

Table 1 synthesizes the aforementioned methods in terms of the size of the graphs they could match, the execution time and the complexity. One can see the complexity of the exact GED in terms of the number of vertices that the methods could match. Based on these facts, researchers shed light on the approximate GED side.

Reference	Size of Graphs	Execution Time	Complexity	Parallel?
(Riesen et al., 2007)	10	10 milliseconds	Exponential	No
(Abu-Aisheh et al., 2015)	15	100000 milliseconds	Exponential	No
(Justice D, 2006)	40	150000 milliseconds	Exponential	No

Table 1: Characteristics of exact graph edit distance methods

3.1.2. 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 (M. Neuhaus and Bunke., 2006). The purpose of BS, is to prune the search tree while searching for an optimal edit path. Instead of exploring all edit paths in the search tree, a parameter s is set to an integer x which is in charge of keeping the x most promising partial edit paths in the set of promising candidates.

In (Riesen, 2009), the problem of graph matching is reduced to finding the minimum assignment cost where in the worst case, the maximum number of operations needed by the algorithm is $O(n^3)$. This algorithm is referred to as BP. Since BP considers local structures rather than global ones, the optimal GED is overestimated. 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 optimal edit distance. In (Riesen and Bunke, 2014), BP is used as an initial step. Then, pairwise swapping of vertices

(local search) is done aiming at improving the accuracy of the distance obtained so far. In (Riesen et al., 2014), a search procedure based on a genetic algorithm is proposed to improve the accuracy of BP. These improvements increase run times. However, they improve the accuracy of the BP solution.

In (Fischer et al., 2015), the authors propose a novel modification of the Hausdorff distance that takes into account not only substitution, but also deletion and insertion cost. $H(V_1, V_2)$ is defined as follows: $H(V_1, V_2) = \sum_{g_1} \min_{g_2} \bar{c_1}(u, v) + \sum_{g_2} \min_{g_1} \bar{c_2}(u, v)$ which can be interpreted as the sum of distances to the most similar vertex in the other graph. This approach allows multiple vertex assignments, consequently, the time complexity is reduced to quadratic (i.e., $O(n^2)$) with respect to the number of vertices of the involved graphs. In (Riesen, 2015; Bougleux et al., 2016), the GED was shown to be equivalent to a Quadratic Assignment Problem (QAP). In (Bougleux et al., 2016), the QAP formulation of the GED problem is solved by two well-known graph matching methods called Integer Projected Fixed Point method (Leordeanu et al., 2009) and Graduated Non Convexity and Concavity Procedure (Liu and Qiao, 2014). These two approximate methods have been adapted and optimized to solve sub-optimally the GED problem. In (Leordeanu et al., 2009), this heuristic improves an initial solution by trying to solve a linear assignment problem and the relaxed QAP where binary constraints are relaxed to the continuous domain. Iteratively, the quadratic formulation is linearly approximated by its 1st-order expansion around the current solution. The resulting assignment helps at guiding the minimization

Reference	Size of Graphs	Execution Time	Complexity	Parallel?
(Riesen, 2009)	100	400 milliseconds	Cubic	No
(Riesen et al., 2007)	70	28 seconds	$\|V_1\|^x$	No
(Fischer et al., 2015)	100	500 milliseconds	Quadratic	No
(Bougleux et al., 2016)	70	27 seconds	Cubic	No

Table 2: Characteristics of approximate GED methods

of the relaxed QAP. In (Liu and Qiao, 2014), a path following algorithm aims at approximating the solution of a QAP by considering a convex-concave relaxation through the modified quadratic function.

3.1.3. Synthesis

Table 2 summarizes the aforementioned approximate GED methods. Approximate GED methods often have a polynomial computational time in the size of the input graphs and thus are much faster than the exact ones. Nevertheless, these methods do not guarantee to find the optimal matching. On the exact GED side, only few approaches have been proposed to postpone the graph size restriction (Tsai et al., 1979; Justice D, 2006; Riesen et al., 2007; Abu-Aisheh et al., 2015). For all these reasons, we believe that proposing a fast and exact GED algorithm is of great interest.

3.2. State-of-the-art of parallel branch-and-bound algorithms

Parallel BnB algorithms have been wildly studied in the past. In this section, we report approaches that have been proposed in the literature to solve BnB in a fully parallel manner. The state-of-the-art in this section is divided into two big families, depending on whether or not the exploration

of the search tree is done in a regular way.

3.2.1. Regular Exploration of the Search Space

Chakroun et al in (Chakroun and Melab, 2013) put forward a template that transforms the unpredictable and irregular workload associated to the explored BnB tree into regular data-parallel GPU kernels [1]. A pool of pending nodes is offloaded to the GPU where each node is evaluated in parallel. Moreover, the branching and pruning steps are performed on the CPU side. In fact, besides equivalent operations, the pruning operator on top of GPU reduces the time of transferring the resulting pool from the GPU to the CPU since the non promising generated sub-problems are kept in the GPU memory and deleted there. The authors of (Boukedjar et al., 2012) presented a CPU-GPU model. When a kernel is launched, the works are assigned to idle threads. Each thread performs computation on only one node of the *BnB* list. Moreover, each thread has its own register and private local memory. Threads can communicate by means of a global memory. Both (Chakroun and Melab, 2013) and (Boukedjar et al., 2012) solved the irregularity of BnB. However, the explorations in both approaches take longer time. That is because in the first kernel each thread generates only one child at each time while the elimination of branches occurs in the second kernel.

An OPEN-MP approaches have been put forward in (Dorta et al., 2003). T threads are established by the master program. Moreover, the master

[1]Kernels are functions executed by many GPU threads in parallel.

program generates tree nodes and put them in a queue. Then T tree nodes are removed from the queue and assigned to each thread. The best solution must be modified carefully where only one thread can change it at any time. The same thing is done when a thread tries to insert a new tree nodes in the global shared queue. In this approach, each thread only takes one node, explores it and at the end of its exploration it sends its result to the master that forwards the message to other slaves if the upper bound is updated. Thus, this model did not tackle the irregularity of BnB.

3.2.2. Irregular Exploration of the Search Space

A *master-slave* parallel formulation of depth-first search was proposed in(Rao and Kumar, 1987). Each thread takes a disjoint part of the search space. Once a thread finishes its assigned part, it steals unexplored nodes of the search space of another thread. A dynamic depth eager scheduling method was proposed in (Neary and Cappello, 2005). In the beginning, a depth parameter is set to 2, which means that all the tasks whose level in the search space is 2 are processed by threads with no further subdivision. Then, each thread works on its associated problems. When a thread runs out of work, it requests work from some thread that it knows. This balances the computational load as long as the number of tasks per thread is high. The communication in (Rao and Kumar, 1987) and (Neary and Cappello, 2005) is asynchronous, and thus threads communicate if they succeed in updating the upper bound. An eager scheduling approach is used to make the tasks

balanced depending on the difficulty of each tree node.

A master-slave hybrid Depth-First/Best-First was proposed in (Chung et al., 2012). The master thread keeps generating the tree nodes at a predetermined level (i.e., level i) and saves them to a work pool. Then, each of the worker threads takes a node with the minimum lower bound from the pool and explores it in a depth-first way. Generating and exploring nodes are repeated until finding the solution of the problem. This model has a less communication between threads, however, it is irregular as the works given to threads do not have the same difficulty.

In (Allen and Yasuda, 1997), a BnB algorithm for solving inexact graph matching was proposed. This algorithm aims at determining a minimum-distance between two unattributed graphs. At each iteration, each thread takes a node from its queue to be solved by expanding it in a depth-first search way until its branch is fully explored, updating the local best permutation and the corresponding degree of mismatch and eliminating test. Afterwards, a global permutation with its corresponding degree of mismatch is updated and given to all threads when all of them finish solving their chosen nodes or problems, then, a next node is chosen by each thread. A thread becomes inactive when it has no node left in its queue. Load balancing is performed if the number of inactive threads with empty queues is above a threshold T. The best permutation and the best degree of match are only updated at the end of each iteration, such a fact will not prune the search space as fast as possible.

3.2.3. Synthesis

Based on the aforementioned parallel BnB algorithms and to the best of our knowledge, none of these algorithms addressed the GED problem. We believe that proposing a parallel branch-and-bound algorithm dedicated to solving the GED problem is of great interest since the computational time will be improved. The search tree of GED is irregular (i.e., the number of tree nodes varies depending on the ability of the lower and upper bounds in pruning the search tree) and thus the regular parallel approaches (e.g., (Chakroun and Melab, 2013; Boukedjar et al., 2012; Dorta et al., 2003)) are not suitable for such a problem.

The approaches in (Rao and Kumar, 1987), (Chung et al., 2012) and (Neary and Cappello, 2005) are interesting since the communication is *asynchronous* [2] and thus there is no need to stop a thread if it did not finish its tasks, unless another thread ran out of tasks. In (Chung et al., 2012), however, load balancing is not integrated. Thus, when there are no more problems to be generated by the master thread, some threads might become idle for a certain amount of time while waiting the other threads to finish their associated tasks. For *GED*, load balancing is important to keep the amount of work balanced between all threads.

On this basis, we propose a parallel GED method equipped with a load balancing strategy. This paper is considered as an extension of the most

[2] Asynchronous communication indicates that no thread waits another thread to finish in order to start its new task (Bertsekas and Tsitsiklis, 1997).

recent BnB algorithm (*DF*). When thinking of a parallel and/or a distributed approach of *DF*, the edit paths can be considered as atomic tasks to be solved. Edit paths can be dispatched and can be given to threads in order to divide the GED problem into smaller problems. It is hard to estimate the time needed by threads to explore a sub-tree (i.e., to become idle). Likewise, the number of CPUs and/or machines have to be adapted to the amount and type of data that have to be analyzed. Some experiments in Section 5.5 illustrate this point and are followed by a discussion.

4. Proposal: Parallel graph edit distance using a load balancing strategy

In this section, an overview of our proposal is given. The main objectives of the approach lie in, first, making sure that all the threads have a work to do. Second, balancing the workload of the threads at run-time. Third, exploring the fruitful partial edit paths of the search tree thanks to the cost estimation of *lb(p)*.

Generally speaking, our proposal is inspired by some ideas in (Rao and Kumar, 1987; Neary and Cappello, 2005). A best-first procedure is performed before starting to decompose the search tree (composed of edit paths) into sub-trees. The load balancing procedure occurs when any thread finishes all its assigned edit paths. The algorithm terminates when all threads finish the exploration of their assigned editpaths. Our algorithm, denoted by *PDFS*, consists of three main steps: Initialization-Decomposition-Assignment, Branch-

and-Bound and Load Balancing. Figure 2 pictures the whole steps of *PDFS*.

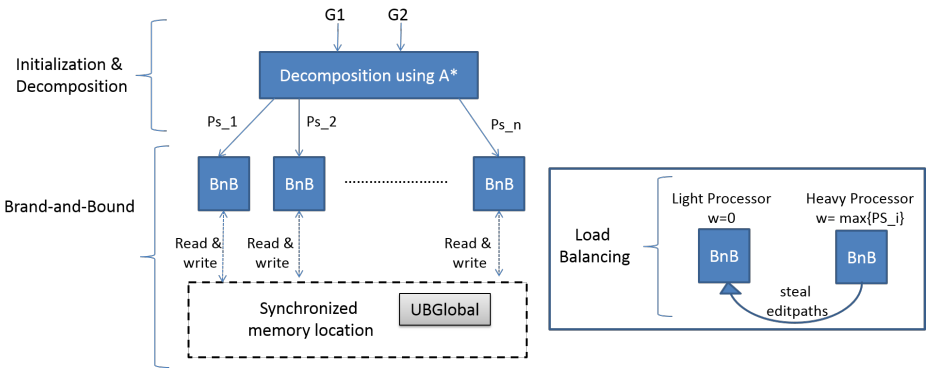


Figure 2: The main Steps of *PDFS*

4.1. Initialization, Decomposition and Assignment

The objective of this step is twofold. First: dividing the problem into sub-problems. Second, making sure, at the beginning of the method, that all threads have an equivalent workload in terms of number of edit paths and their difficulty.

Initialization Procedure. As in *DF* (Abu-Aisheh et al., 2015), this phase consists of 3 steps, each of which aims at speeding up the calculations.

First, the vertices and edges cost matrices (C_v and C_e) are constructed, respectively. This step aims at getting rid of re-calculating the distances between attributes when matching vertices and edges of G_1 and G_2.

Let $G_1 = (V_1, E_1, \mu_1, \xi_1)$ and $G_2 = (V_2, E_2, \mu_2, \xi_2)$ be two graphs with $V_1 = (u_1, ..., u_n)$ and $V_2 = (v_1, ..., v_m)$. A vertices cost matrix C_v, whose dimension is $(n + 2)$ X $(m + 2)$, is constructed as follows:

$$C_v = \left|\begin{array}{ccc||ccc} c_{1,1} & \ldots & c_{1,\kappa} & c_{1\leftarrow\epsilon} & \ldots & \infty \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ c_{n,1} & \ldots & c_{n,\kappa} & \infty & \ldots & c_{n\rightarrow\epsilon} \\ \hline\hline c_{\epsilon\rightarrow 1} & \ldots & \infty & \infty & \ldots & \infty \\ \infty & \ldots & c_{\epsilon\leftarrow\kappa} & \infty & \ldots & \infty \end{array}\right|$$

where n is the number of vertices of G_1 and κ is the number of vertices of G_2 .

Each element $c_{i,j}$ in the matrix C_v corresponds to the cost of assigning the i^{th} vertex of the graph G_1 to the j^{th} vertex of the graph G_2. The left upper corner of the matrix contains all possible node substitutions while the right upper corner represents the cost of all possible vertices insertions and deletions of vertices of G_1, respectively. The left bottom corner contains all possible vertices insertions and deletions of vertices of G_2, respectively whereas the bottom right corner elements cost is set to infinity which concerns the substitution of $\epsilon - \epsilon$. Similarly, C_e contains all the possible substitutions, deletions and insertions of edges of G_1 and G_2. C_e is constructed in the very same way as C_v. The aforementioned matrices C_v and C_e are used as an input of the following phase.

Second, the vertices of G_1 are sorted in order to start with the most promising vertices in G_1. BP is applied to establish the initial edit path EP (Riesen, 2009). Afterwards, the edit operations of EP are sorted in ascending

order of the matching cost where $EP_{sorted} = \{u \rightarrow v\}\ \forall u \in V_1 \cup \{\epsilon\}$. At last, from EP_{sorted}, each $u \in V_1$ is inserted in *sorted*-V_1. This operation helps in finding the most promising vertices $vi \in V_1$ that will be matched first with the unmatched vertices in V_2 to speed up the exploration of the search tree while searching for the optimal solution.

Third, a first upper bound (UB) is computed by BP algorithm as it is relatively fast and it provides reasonable results, see (Riesen, 2009) for more details.

Decomposition. Before starting the parallelism, a distribution approach is applied aiming at dispatching the workload or sub-problems among threads. For that purpose, N edit paths are first generated using A^* by the main thread and saved in the heap. Afterwards, the N partial edit paths are sorted as an ordered tree starting from the node whose $lb(p)$ is minimum up to the most expensive one. Note that N is a parameter of *PDFS*.

Assignment. Let $\mathcal{Q}$ be the set of partial solutions outputted by A^*. Assigning partial solutions to parallel threads is equivalent to solving the static load balancing problem stated in Section 2.3.1. Due to the complexity of the problem, we chose to avoid an exact computation of load balancing and we adopted an approximated one. Algorithm 1 depicts the strategy we have followed. Once the partial edit paths are sorted in the centralized heap (line 1), the local list *OPEN* of each thread is initialized as an empty set (lines 2 to 4). Each thread receives one partial solution at a time, starting from

the most promising partial edit paths (line 8). The threads keep taking edit paths in that way until there is no more edit path in the centralized heap (lines 6 to 10).

Algorithm 1 Dispatch-Tasks

Input: A set of partial edit paths $\mathcal{Q}$ generated by A^* and T threads.
Output: The local list $OPEN$ of each thread T_i

1: $\mathcal{Q} \leftarrow$ sortAscending($\mathcal{Q}$)
2: **for** $T_{index} \in T$ **do**
3: $\quad OPEN_{T_{index}} \leftarrow \{\phi\}$
4: **end for**
5: i=0 ▷ a variable used for thread's indices
6: **for** $p \in \mathcal{Q}$ **do**
7: $\quad index = i \ \% \ |T|$
8: $\quad OPEN_{T_{index}}$.addTask(p)
9: $\quad i{+}{+}$
10: **end for**
11: Return $OPEN_{T_{index}} \quad \forall\, index \in 1, \cdots, |T|$

Each thread maintains a local heap to keep the assigned edit paths for exploring edit paths locally. Such an iterative way guarantees the diversity of nodes difficulty that are associated to each thread.

4.2. Branch-and-Bound Method

In this section we explain the components of BnB that each thread executes on its assigned partial edit paths. First, the rules of selecting edit paths, branching and bounding are described. Second, updating the upper bound and pruning the search tree are detailed.

Selection Rule. A systematic evaluation of all possible solutions is performed without explicitly evaluating all of them. The solution space is organized

as an ordered tree which is explored in a depth-first way. In depth-first search, each edit path is visited just before its children. In other words, when traversing the search tree, one should travel as deep as possible from node i to node j before backtracking. At each step, the most promising child is chosen.

Branching Procedure. Initially each thread only has its assigned editpaths in its local heap set ($OPEN$) i.e., the set of the edit paths, found so far. The exploration starts with the first most promising vertex u_1 in *sorted*-V_1 in order to generate the children of the selected editpath. The children consist of substituting u_1 with all the vertices of G_2, in addition to the deletion of u_1 (i.e., $u_1 \Rightarrow \epsilon$). Then, the children are added to $OPEN$. Consequently, a minimum edit path (p_{min}) is chosen to be explored by selecting the minimum cost node (i.e., $min(g(p) + h(p))$) among the children of p_{min} and so on.

Starting from the second promising vertex u_2 in *sorted*-V_1, the edges of both G_1 and G_2 are handled. Edges of G_1 can be either matched with edges of G_2 or deleted while edges of G_2 can be either inserted in G_1 or matched with edges in G_1. However, the decision of whether an edge is inserted, substituted, or deleted is done regarding the matching of their adjacent vertices. That is, the neighborhood of edges dominates their matchings. Edges are handled as follows: Let u_i and $u_j \in V_1$ be matched with v_k and $v_z \in V_2$, respectively (i.e., $u_i \rightarrow v_k$ and $u_j \rightarrow v_z$). Based on these matchings. One of the following edge operations is selected:

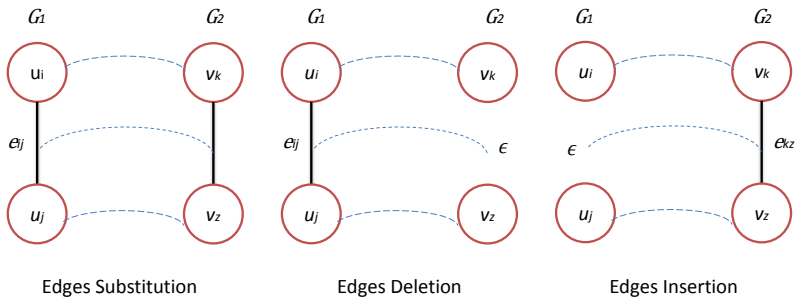


Figure 3: Edge mappings based on their adjacent vertices and whether or not an edge between two vertices can be found

- If $\exists e_{ij} \in E_1$ and $\exists e_{kz} \in E_2$ then $e_{ij} \rightarrow e_{kz}$
- If $\exists e_{ij} \in E_1$ and $\nexists e_{kz} \in E_2$ then $e_{ij} \rightarrow \epsilon$
- If $\nexists e_{ij} \in E_1$ and $\exists e_{kz} \in E_2$ then $\epsilon \rightarrow e_{kz}$

The search for a better edit path continues through backtracking if p_{min} equals ϕ. In this case, the next child of p_{min} is tried out and so on.

Pruning Procedure. As in *DF*, pruning, or bounding, is achieved thanks to $h(p)$, $g(p)$ and an upper bound UB obtained at node leaves. Formally, for a node p in the search tree, the sum $g(p) + h(p)$ is taken into account and compared with UB. That is, if $g(p) + h(p)$ is less than UB then p can be explored. Otherwise, the encountered p will be pruned from *OPEN* and a backtracking is done looking for the next promising node and so on until finding the best UB that represents the optimal solution of *PDFS*. Note that *OPEN* is a local search tree of each thread. This algorithm differs from A^* as at any time t, in the worst case, *OPEN* contains exactly $|V_1|.|V_2|$ elements and hence the memory consumption is not exhausted.

Upper Bound Update. The best upper bound is globally shared by all threads (shared UB). When a thread finds a better upper bound, the shared UB is updated (*i.e.*, a complete path found by a thread whose cost is less than the current UB).

Heuristic. After comparing several heuristics *h(p)* from the literature, we selected the bipartite graph matching heuristic proposed in (Riesen, 2009). The complexity of such a method is $O(\{|V_1|, |V_2|\}^3 + |E_1|, |E_2|\}^3)$. For each tree node p, the unmatched vertices and edges are handled in a complete independent way. Unmatched vertices of G_1 and unmatched vertices of G_2 are matched at best by solving a linear sum assignment problem. Unmatched edges of both graphs are handled analogously. Obviously, this procedure allows multiple substitutions involving the same vertex or edge and, therefore, it possibly represents an invalid way to edit the remaining part of G_1 into the remaining part of G_2. However, the estimated cost certainly constitutes a lower bound of the exact cost.

4.3. Load Balancing and Communication

Load Measurement. Each thread i provides some information about its workload or weight index ω_i. Obviously, the number of edit paths in *OPEN* can be a workload index. However, this choice may not be accurate since BnB computations are irregular with different computational requirements. Several workload indices can be adapted. One could think about $h(p)$. $h(p)$ can be hard to interpret, it can be small either because the p is close to

the leaf node or because p is a very promising solution. To eliminate this ambiguity, instead, one can count the number of vertices in G_1 that have not been matched yet. This is done on each edit path in the local heap. In our approach, we have selected the latter.

Initiation Rule. An initiation rule dictates when to initiate a load balancing operation. Its invocation decision must appear when a thread workload index ω_i reaches a zero value that is to say if the thread is *idle*.

Load Balancing Operation. In parallel BnB computations, each process solves one or more subproblems depending on the decomposite procedure. In our problem, two threads are involved in the load balancing operation: Heavy and idle threads. When a thread becomes idle, the heaviest thread will be in charge of giving to the idle thread some edit paths to explore. All the edit paths of the heavy thread are ordered using their $lb(p)$. The heavy thread distributes the best edit paths between it and the idle thread. This procedure guarantees the exploration of the best edit paths first since each thread holds some promising edit paths.

Threads Communication. All threads share C_e, C_e, *sorted*-V_1 and *UB*. Since all threads try to find a better *UB*, a memory coherence protocol is required on the shared memory location of *UB*. When two threads simultaneously try to update *UB*, a synchronization process based on mutex is applied in order to make sure that only one thread can access the resource at a given point in time.

5. Experiments

This section aims at evaluating the proposed contribution through an experimental study that compares 9 methods in terms of precision, execution time and classification rate on reference datasets. We first describe the datasets, the methods that have been studied and the protocol. Then, the results are presented and discussed.

5.1. Datasets

To the best of our knowledge, few publicly available graphs databases are dedicated to precise evaluation of graph matching tasks. However, most of these datasets consist of synthetic graphs that are not representative of PR problems concerning graph matching under noise and distortion. We shed light on the IAM graph repository which is a widely used repository dedicated to a wide spectrum of tasks in pattern recognition and machine learning (Riesen, 2008). Moreover, it contains graphs of both symbolic and numeric attributes which is not often the case of other datasets. Consequently, the GED algorithms involved in the experiments are applied to three different real world graph datasets taken from the IAM repository (Riesen, 2008) (i.e., GREC, Mutagenicity (MUTA) and Protein datasets). Continuous attributes on vertices and edges of GREC play an important role in the matching process whereas MUTA is representative of GM problems where graphs have only symbolic attributes. On the other hand, the Protein database contains numeric attributes on each vertex as well as a string sequence that is used to

represent the amino acid sequence. For the scalability experiment, the subsets of GREC, MUTA and Protein, proposed in the repository GDR4GED (Abu-Aisheh et al., 2015), were chosen. On the other hand, for the classification experiment, the experiments were conducted on the train and test sets of each of them.

In addition to these datasets, a chemical dataset, called PAH, taken from GREYCs Chemistry dataset repository [3], was also integrated in the experiments. This dataset is quite challenging since it has no attributes on both vertices and edges. Table 3 summarizes the characteristics of all the selected datasets.

These datasets have been chosen by carefully reviewing all the publicly available datasets that have been used in the reference works mentioned in section 3 (LETTER, GREC, COIL, Alkane, FINGERPRINT, PAH, MUTA, PROTEIN and AIDS to name the most frequent ones). On the basis of this review, a subset of these datasets has been chosen in order to get a good representativeness of the different graph features which can affect GED computation (size and labelling):

Each dataset has specific edit cost functions. Two non-negative meta parameters are associated to GM: (τ_{vertex} and τ_{edge}) where τ_{vertex} denotes a vertex deletion or insertion costs whereas τ_{edge} denotes an edge deletion or insertion costs. A third meta parameter α is integrated to control whether

[3]https://brunl01.users.greyc.fr/CHEMISTRY/index.html

Dataset	GREC	Mutagenicity	Protein	PAH
Size	4337	1100	660	484
Vertex labels	x,y coordinates	Chemical symbol	Type and amino acid sequence	None
Edge labels	Line type	Valence	Type and length	None
$\overline{vertices}$	11.5	30.3	32.6	20.7
$\overline{edges}$	12.2	30.8	62.1	24.4
Max vertices	25	417	126	28
Max edges	30	112	146	34

Table 3: The Characteristics of the GREC, Mutagenicity, Protein and PAH Datasets.

the edit operation cost on the vertices or on the edges is more important. Table 4 demonstrates the cost functions of each of the included datasets as well as their meta parameters.

Dataset	GREC	Mutagenicity	Protein	PAH
τ_{vertex}	90	11	11	3
τ_{edge}	15	1.1	1	3
α	0.5	0.25	0.75	0.5
Vertex substitution function	Extended euclidean distance	Dirac function	Extended string edit distance	0
Edge substitution function	Dirac function	Dirac function	Dirac function	0
Reference of cost functions	(Riesen, 2009)	(Riesen, 2009)	(Riesen, 2009)	(Gauzere et al., 2012)

Table 4: The cost functions and meta parameters of the datasets.

5.2. Studied Methods

We compared *PDFS* to five other GED algorithms from the literature. From the related work, we chose two exact methods and three approximate methods. On the exact method side, A* algorithm applied to GED problem (Riesen et al., 2007) is a foundation work. It is the most well-known exact method and it is often used to evaluate the accuracy of approximate methods. *DF* is also a depth-first GED that has been recently published and that beats A^* in terms of running time and precision (Abu-Aisheh et al., 2015).

Moreover, a naive parallel *PDFS*, referred to as *naive-PDFS*, is implemented and added to the list of exact methods. The basis of *naive-PDFS* is similar to *PDFS*. However, *naive-PDFS* does not include neither the assignment phase (see Section 4.1) nor the load balancing phase (see Section 4.3). Instead of the assignment phase, a random assignment is applied. *naive-PDFS* does not include a balancing strategy which means that if a thread T_i finished its assigned nodes, it would be idle during the rest of the execution of *naive-PDFS*. On the approximate method side, we can distinguish three families of methods, tree-based methods, assignment-based methods and set-based methods. For the tree-based methods, the truncated version of A* (i.e., *BS-*x) was chosen where x refers to maximum number of open edit paths. Among the assignment-based methods, we selected BP. In (Riesen, 2009), authors demonstrated that BP is a good compromise between speed and accuracy. Finally, we picked a set-based method. An approach based on the Hausdorff matching, denoted by H, was proposed in (Fischer et al., 2015). All these methods cover a good range of GED solvers and return a vertex to vertex matching, except H, as well as a distance between two graphs G_1 and G_2 except the lower bound GED which only returns a distance between two graphs.

5.3. Environment

PDFS and *naive-PDFS* were implemented using Java threads. The evaluation of both algorithms was conducted on a 24-core Intel i5 processor

2.10GHz, 16GB memory. In *PDFS*, the partial edit paths are sorted in the centralized heap *OPEN*, as mentioned in Section 4.1. Each thread takes one edit path from *OPEN* and the edit path is then deleted from *OPEN*. The CPU only needs to move to the next memory location so the spatial locality is exploited at best to reduce cache misses. For sequential algorithms, evaluations were conducted on one core.

5.4. Protocol

In this section, the experimental protocol is presented and the objectives of the experiment are described.

Let $\mathcal{S}$ be a graph dataset consisting of k graphs, $\mathcal{S} = \{g_1, g_2, ..., g_k\}$. Let $\mathcal{M} = \mathcal{M}_e \cup \mathcal{M}_a$ be the set of all the GED methods listed in Section 5.2, with $\mathcal{M}_e = \{A*, DF, PDFS\}$ the set of exact methods and $\mathcal{M}_a = \{BP$,*BS-1*,*BS-10*,*BS-100*$, H\}$ the set of approximate methods (where x in BS was set to 1, 10 and 100). Given a method $m \in \mathcal{M}$, we computed all the pairwise comparisons $d(g_i, g_j)^m$, where $d(g_i, g_j)^m$ is the value returned by method m on the graph pair (g_i, g_j) within certain time and memory limits.

Two types of experiments were carried out scalability experiment and classication experiment.

5.4.1. Scalability Experiment under Time Constraints

In the scalability experiment, several metrics were included: The number of best found solutions and the number of optimal solutions. Moreover, a projection of p on a two-dimensional space ($\mathbb{R}^2$) is achieved by using *speed-score*

and *deviation-score* features where speed and deviation are two concurrent criteria to be minimized. First, for each database, the mean deviation and the mean time is derived as follows:

$$\overline{dev_k^p} = \frac{1}{m \times m} \sum_{i=1}^{m} \sum_{j=1}^{m} dev(g_i, g_j)^p \quad \forall p \in P \quad \forall k \in \#subsets \tag{1}$$

$$\overline{time_k^p} = \frac{1}{m \times m} \sum_{i=1}^{m} \sum_{j=1}^{m} time(G_i, G_j)^p \; and \; (i,j) \in [\![1,m]\!]^2 \quad \forall k \in \#subsets \tag{2}$$

where $dev(G_i, G_j)$ is the deviation of each $d(G_i, G_j)$ and $time(G_i, G_j)$ is the run time of each $d(G_i, G_j)$. To obtain comparable results between databases, mean deviations and times are normalized between 0 and 1 as follows:

$$\overline{deviation\_score^m} = \frac{1}{\#subsets} \sum_{\mathcal{S} \in subsets} \frac{\overline{dev_{\mathcal{S}}^m}}{max\_dev_{\mathcal{S}}} \tag{3}$$

$$\overline{time\_score^m} = \frac{1}{\#subsets} \sum_{\mathcal{S} \in subsets} \frac{\overline{time_{\mathcal{S}}^m}}{max\_time_{\mathcal{S}}} \tag{4}$$

where $max\_dev_{\mathcal{S}}$ and $max\_time_{\mathcal{S}}$ denote respectively the maximal mean deviation and the maximal mean execution time obtained among all the methods $\mathcal{M}$ on dataset $\mathcal{S}$.

All these metrics have been proposed in (Abu-Aisheh et al., 2015). This

experiment was decomposed of 2 tests:

Accuracy Test. The aim was to illustrate the error committed by approximated methods over exact methods. In an ideal case, no time constraint (C_T) should be imposed to reach the optimal solution. Due to the large number of considered matchings and the exponential complexity of the tested algorithms, we allowed a maximum C_T of **300 seconds**. This time constraint was large enough to let the methods search deeply into the solution space and to ensure that many nodes will be explored. The key idea was to reach the optimality, whenever it is possible, or at least to get to the *Graal* (i.e., the optimal solution) as close as possible. This use case is necessary when it is important to accurately compare images represented by graphs even if the execution time is long.

Speed Test. The goal was to evaluate the accuracy of exact methods against approximate methods when time matters. That is to say in a context of very limited time. Thus, for each dataset, we select the slowest graph comparison using an approximate method among BP and H as a first time constraint. Unlike BP and H, BS is not included as it is a tree-search algorithm which could output a solution even under a limited C_T. Mathematically saying, C_T is defined as follows:

$$C_T = \max_{m,i,j}\{time_m(g_i, g_j)\} \tag{5}$$

Where $m \in \mathcal{M}_s$ $/BS$, $(i,j) \in [\![1,k]\!]^2$ and $time$ is a function returning the running time of method m for a given graph comparison. This way ensures that BP and H could solve any instance. When the time limit is over, the best solution found so far is outputted by BS as well as the exact GED methods. So time and memory limits play a crucial role in our experiments since they impact such methods. In Table 5, we display the time limits used for each dataset.

Dataset	**GREC**	**MUTA**	**Protein**	**PAH**
C_T (milliseconds)	400	500	400	55

Table 5: Time constraints for accuracy evaluation in limited time

This case study is representative of a classification stage where many distances have to be quickly computed.

5.4.2. Classification experiment

This part of the experiments aimed at showing the performance of the included methods in classifying the graphs of the test set of each of GREC, MUTA and Protein. PAH is not included since we do not have the classes of the test graphs.

Two metrics are proposed: Average time (i.e., the time needed to classify each test graph) and classification rate using 1 nearest neighbor (1-NN). The values of C_T were the same ones used in the speed test of the aforementioned experiment (speed test).

In all the experiments (i.e., scalability and classification), C_M was set to 1GB. Among all the aforementioned methods, we expected A^* to violate

C_M specially when graphs get larger. In a small C_T context, the number of threads in *PDFS* was set to 3. The reason is that since C_T was quite small, we did not want to lose time decomposing the workload among a big number of threads. Moreover, because of the complexity of the calculation of lb, it was removed from each of *BS*, A^*, *DF* and $PDFS$

5.5. Parameters

We study the effect of increasing the number of threads T on both accuracy and speed of *naive-PDFS* and *PDFS*. This test was carried out using a 24-core CPU. T is varied from 2 to 128 threads. Moreover, the effect of several values of N, described in Section 4.1, were studied. Five values of N were chosen: -1, 100, 250, 500 and 1000, where N=-1 represents the decomposition of the first floor in the search tree with all possible branches, N= 100 and 250 moderately perform load balancing while N=500 and 1000 is the exhaustive case where threads have much less time dedicated to load balancing since each thread will be assigned sufficient number of works before the parallelism starts. We expected *PDFS* to perform better when increasing N up to a threshold where the accuracy of the algorithm is degraded.

5.6. Results

In this section, the results are demonstrated along with their discussions. We conducted experiments on the involved datasets, however, for the part of parameters selection, we only show the results on GREC-20 (Abu-Aisheh

et al., 2015) since this dataset is representative of the other datasets. Time unit is always expressed in milliseconds.

5.6.1. Number of Threads

Table 6 displays the effect of the number of threads $|T|$ on the performance of *PDFS*. In Table 6, CPU time is the time spent at working by all the threads. One may notice that increasing $|T|$ resulted in increasing the chance to find a better solution, more optimal solutions and a smaller deviation as we explored more nodes in a parallel manner. Thus, the overall running time decreased (see Figure 4). Since the machine on which we ran this test has a 24-core processor, there was a saturation when increasing $|T|$. For example, on 128 threads the deviation became bigger (see Figure 4). On a 24-core machine, 32 and 64 threads had got the best results. In addition, increasing the number of threads also increased the load balancing.

Method	**#best found solutions**	**#optimal solutions**	**Idle Time over CPU Time**
PDFS-2T	67	48	$1.7 * 10^{-5}$
PDFS-4T	79	54	$9.5 * 10^{-5}$
PDFS-8T	83	66	$2.8 * 10^{-4}$
PDFS-16T	92	69	$7.7 * 10^{-4}$
PDFS-32T	94	69	0.011
PDFS-64T	95	68	0.043
PDFS-128T	98	66	0.169

Table 6: The effect of the number of threads on the performance of *PDFS*.

Based on the aforementioned results, $|T|$ is set to 64. For *naive-PDFS*, the same experiment was conducted. At the end, $|T|$ was set to 128.

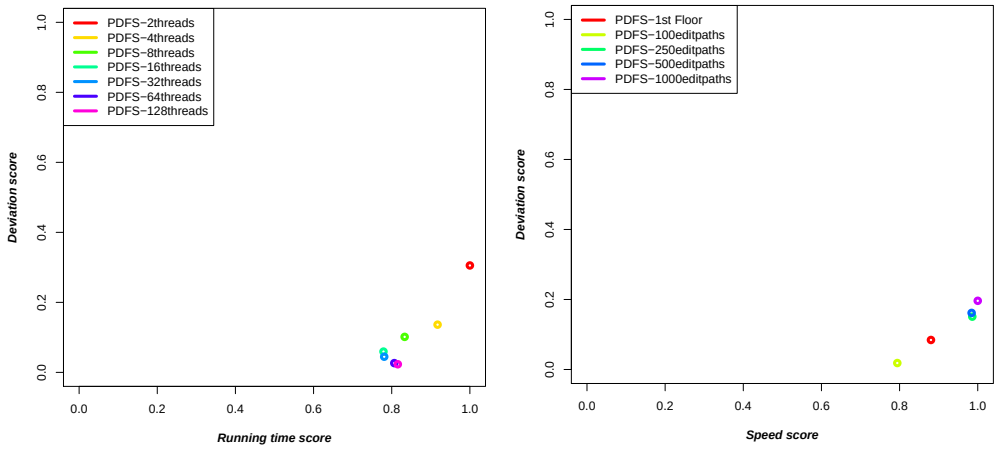


Figure 4: Time-deviation score: Left (#Threads), right (# Edit Paths).

5.6.2. Number of Edit Paths

Table 7 demonstrates the effect of the number of initial edit paths N on the performance of *PDFS*. One can remark that N equals 100 was the best choice in terms of the number of best found solutions, number of optimal solutions and deviation. Even though N equals 100 remarkably spent much more time on load balancing, it was still 2.3 times more precise than N equals 1000. The latter represented the least precise results (see Figure 4) which was due to the time spent in dispatching the work among threads before the BnB step started. In the rest of the experiments, the number of initial edit paths is set to 100.

For *naive-PDFS*, N equals 100 also demonstrated the best results.

5.6.3. Comparing PDFS *with* naive-PDFS

In this section we compare both *PDFS* and *naive-PDFS*. For comparison needs, both algorithms were executed on 128 threads which is slightly in

Method	#best found solutions	#optimal solutions	Idle Time over CPU Time
PDFS-1st Floor	82	53	0.067
PDFS-100 EP	85	66	0.067
PDFS-250 EP	85	41	0.053
PDFS-500 EP	82	41	0.023
PDFS-1000 EP	83	40	0.018

Table 7: The effect of the number of edit paths on the performance of *PDFS*

favor of *naive-PDFS*.

The results in Table 8 show that *PDFS* beat *naive-PDFS* with 28 more optimal solutions. PDFS is equipped with a load balancing scheme which allows the workload variance to be minimized. The workload variance is defined as the deviation between the threads's workloads and the average workload of all threads at time t. Reducing the variance is important to make sure that all threads have approximately the same amount of work. One can also observe that *PDFS* was fully parallel where the CPU time was doubled compared to *naive-PDFS*. In fact, in *naive-PDFS*, some threads became idle since they finished they assigned works while other threads continued to explore their assigned edit paths.

Method	#optimal solutions	Mean CPU Time (ms)	Mean Variance (ms)
naive-PDFS	63	4792444	361157.6
PDFS	91	7275476	49880.62

Table 8: The effect of the number of edit paths on the performance of *PDFS* when executed on the GREC dataset

5.6.4. Comparing Methods under Constraints

In this section, we compare the state-of-the-art methods as well as *PDFS* under small and large time constraints.

Large Time Constraint. Regarding the number of best found solutions and the number of optimal solutions, *PDFS* always outperformed *DF* on GREC, MUTA, Protein and PAH, see Figures 5, 6 and 7.

On MUTA, the deviation of *BP* was 20%; this fact confirms that the more complex the graphs the less accurate the answer achieved by *BP*, see Figure 7(b). *BP* considers only local, rather than global, edge structure during the optimization process (Riesen, 2009) and so when graphs get larger, its solution becomes far from the exact one. Despite the out-performance of *PDFS* over *BP*, *H* and *DF*, it did not outperform *BS* in terms of number of best found solutions, see Figure 5(b). The major differences between these algorithms are the search space and the Vertices-Sorting strategies which are adapted in *PDFS* and not in *BS*. Since *BP* did not give a good estimation on MUTA, it was irrelevant when sorting 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. However, the difference between *BS* and *PDFS* in terms of deviation was only 0.1%.

On Protein-30, BS-100 was superior to *PDFS* in terms of number of best found solutions with 50 better solutions. However, this is not the case of a bigger dataset like Protein-40 where *BS-100* outputted unfeasible solutions because of the tremendous size of the search tree and thus *PDFS* outperformed it. On average, on all databases and among all methods, *PDFS* got the best deviation, see Figure 7.

Exploring the search tree in a parallel way has an advantage when we are

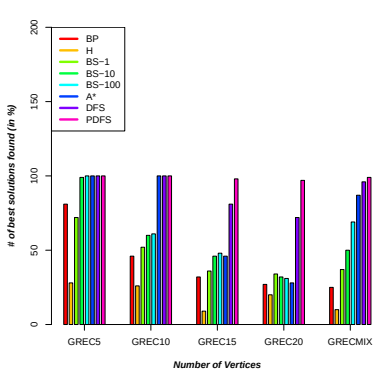
also interested in having more optimal solutions, see Figure 6. Results, in Figure 6, demonstrated that the number of optimal solutions found by *PDFS* was always equal or greater than the number of optimal solutions found by *DF* and A^*, except on MUTA-20 where A^* outperformed it. For instance, on GREC, *PDFS* found 9.6% more optimal solutions when compared to *DF* and 10% more optimal solutions on PAH. Note that without time constraints all the exact GED algorithms must find all the optimal solutions except A^* that has memory bottleneck.

Small Time Constraint. Concerning the number of best found solutions, even under a small C_T, *PDFS* outperformed *DF* where the average difference between *DF* and *PDFS* was: 10% on GREC, 16% on MUTA, 15% on Protein and 11% on PAH, see Figure 8.

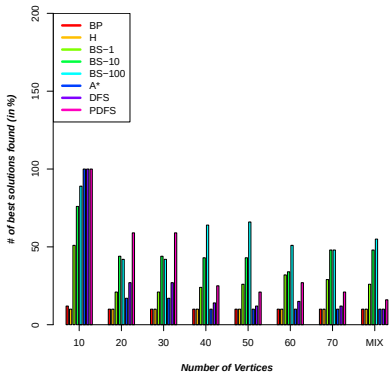
A^* got the highest deviation rates (around 30% on GREC, 73% on MUTA, 86% on Protein and 51.94% on PAH) since it did not have time to output feasible solutions. Despite the fact that *PDFS* was among the slowest algorithms, it obtained the lowest deviation (0% on both GREC and Protein, 5% on MUTA and 6% on PAH), see Figure 9. *BS-100* outputted unfeasible solutions on MUTA-50, MUTA-60, MUTA-70, MUTA-MIX and Protein due to the small C_T.

5.6.5. Classification Tests

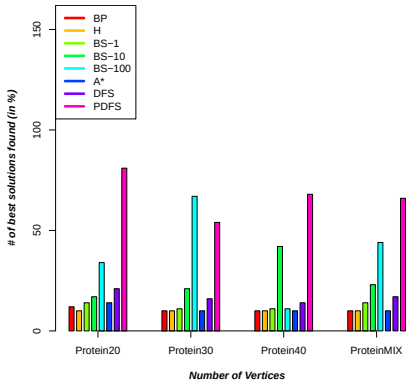
Table 9 shows the methods included in the classification experiments. Different versions of *DF* and A^* were tested on each dataset taking into ac-



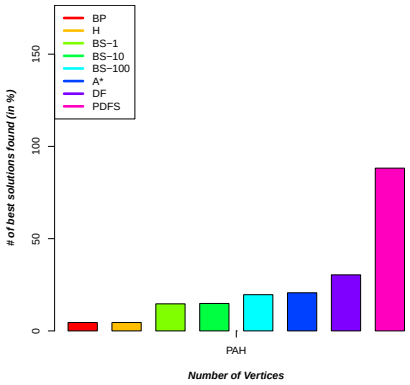
(a) GREC



(b) MUTA

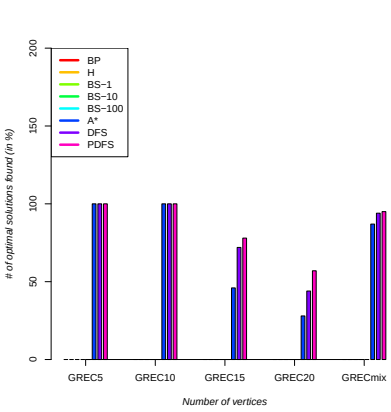


(c) Protein

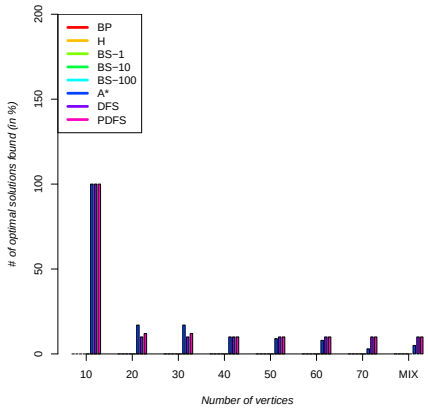


(d) PAH

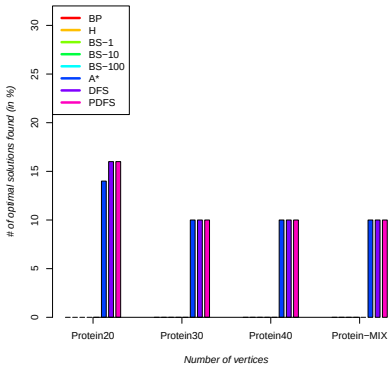
Figure 5: Number of best found solution under big time constraint



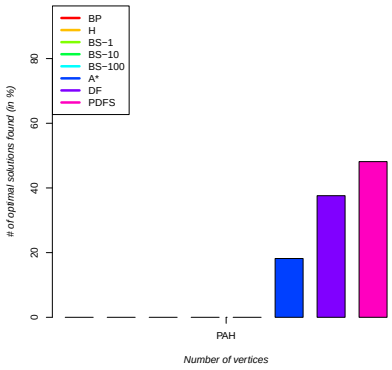
(a) GREC



(b) MUTA

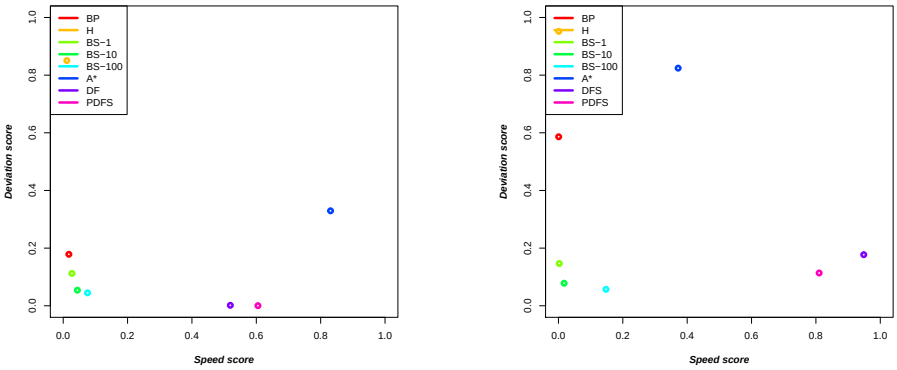


(c) Protein



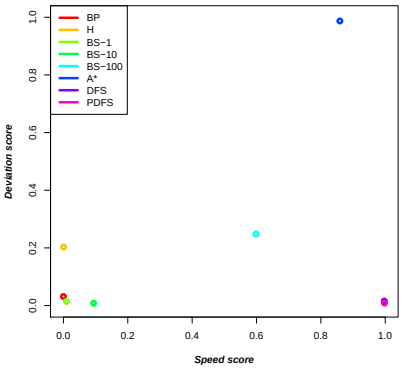
(d) PAH

Figure 6: Number of optimal solutions under big time constraint

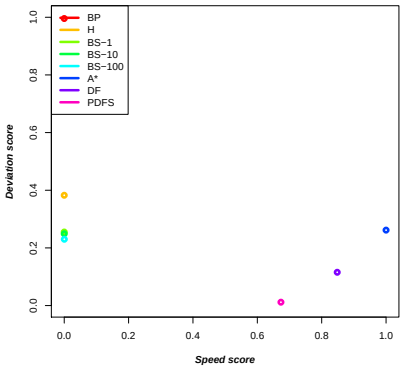


(a) GREC

(b) MUTA

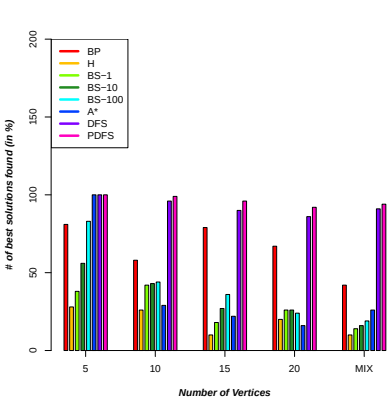


(c) Protein

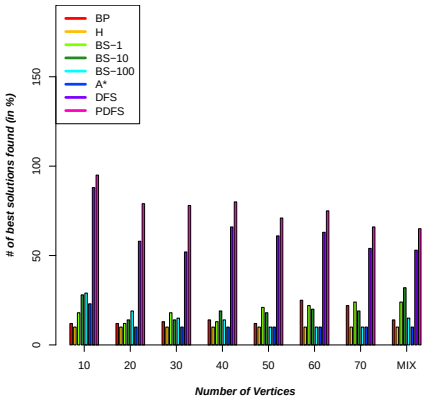


(d) PAH

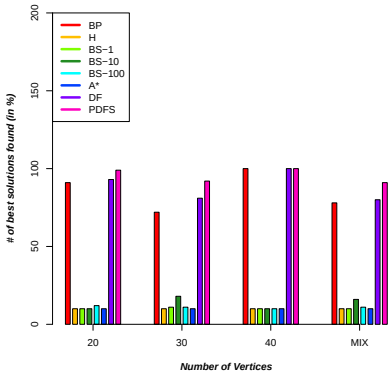
Figure 7: Time-deviation score under large time constraint



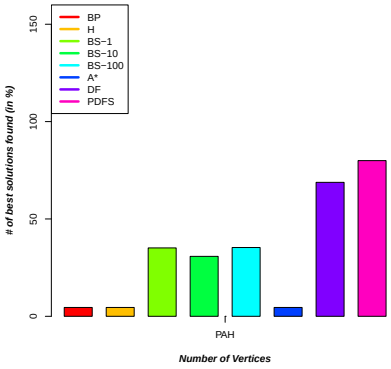
(a) GREC



(b) MUTA

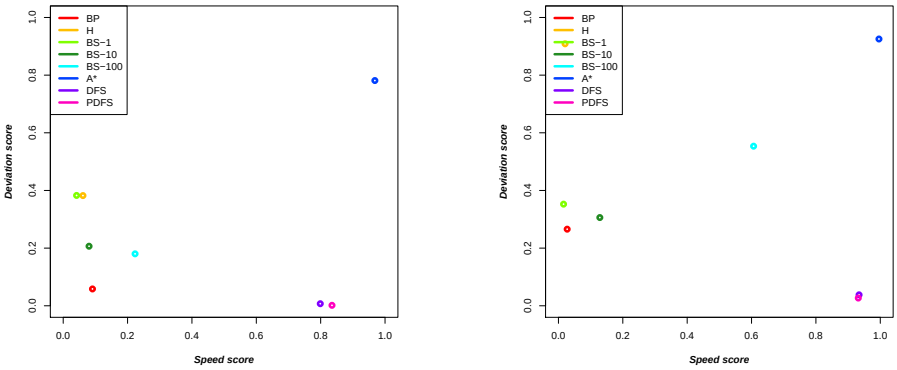


(c) Protein



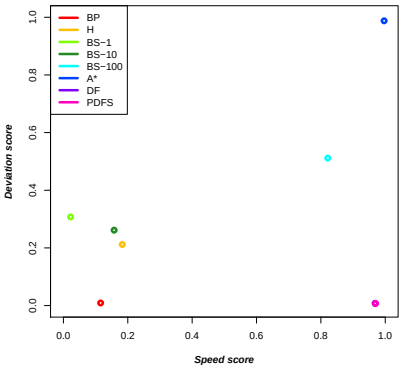
(d) PAH

Figure 8: Number of best found solution under small time Constraint

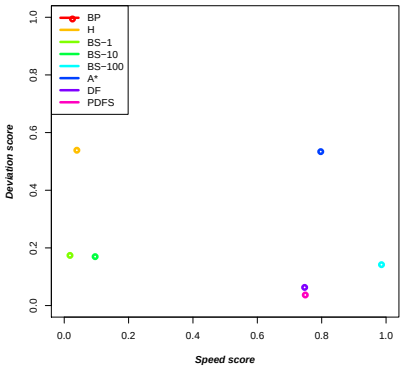


(a) GREC

(b) MUTA



(c) Protein



(d) PAH

Figure 9: Time-deviation score under small time constraint.

count different combinations of lb and UB. Afterwards, the best combination is selected to be compared with the other methods.

Acronym	**Details**
DF-$\overline{UB}$*-*$\overline{LB}$	*DF* without upper bound and with h(p)=0.
DF-$\overline{UB}$*-LB*	*DF* without *UB* and with h(p)=*lb2*.
DF-UB-$\overline{LB}$	*DF* with an initial *UB* equals to *BP*, h(p)=0.
DF-UB-LB	*DF* with an initial *UB* equals to *BP* and *lb2*
PDFS	Parallel GED with the best parameters of *DF*
A^**-*$\overline{LB}$	the A^* algorithm with *lb2*
A^*	the A^* algorithm without *lb2*
BS-1, *BS-10* and *BS-100*	Beam Search with *OPEN* size = 1, 10 and 100, respectively
BP	The bipartite GM
H	The hausdorff algorithm.

Table 9: Methods included in the classification experiments

Table 10 shows the classification results on GREC and Protein. On GREC, *DF* with all its variants obtained the same classification rate as *BP* (i.e., 0.985) even the one without upper and lower bounds (i.e., *DF-*$\overline{UB}$*-*$\overline{LB}$). That shows that *DF* can also be used to classify graphs even without being obliged to wait for the final, or optimal, solution. *DF-UB-LB* was the fastest compared to all the variants. This fact shows the importance of *UB* and *LB* to make the algorithm faster. Accordingly, and since *PDFS* is an extension of *DF*, not all the variants of *PDFS* are tested. That is, only *PDFS-UB-LB* has been included in the tests. *PDFS-UB-LB* was 29% faster than *DF-UB-*

LB. Despite the fact that H was the worst algorithm when evaluating its distances, it was among the algorithms whose classification rate were high. One can see that, on GREC, H beat both *BS-10* and *BS-100*. A^*-$\overline{LB}$ obtained better classification rate than A^*. A^*' lower bound is time consuming and consequently the number of unfeasible solutions was high.

On Protein, one can see a different behavior (see Table 10). *DF-UB-*$\overline{LB}$ was the fastest while *DF-UB-LB* was the slowest. That is because of the time consumed to calculate distances using the cost functions of Protein. Thus, as on GREC, *PDFS-UB-*$\overline{LB}$ was included in the tests. Despite the slowness of *DF-UB-LB*, it was also the best algorithm in terms of classification rate. *PDFS-UB-*$\overline{LB}$ was 36% faster than *DF-UB-*$\overline{LB}$. Even though *BS* took relatively enough time to classify graphs (compared to *DF*), it was way far from the results obtained by *DF*. A^* was not able to find feasible solutions of each pair of graphs. That was not the case of all the variants of *DF* as they were always able to output feasible solutions before halting.

Computing *lb(p)* and a first upper bound *UB* was time consuming on such a large database. Since C_T of MUTA was set to 500ms, we kept only *DF-*$\overline{UB}$*-*$\overline{LB}$ and A^*-$\overline{LB}$. Results showed that *DF-*$\overline{UB}$*-*$\overline{LB}$ was twice as slow as *BP*, however, both of them succeeded in finding the best classification rate (i.e., approximately 0.70). *PDFS-*$\overline{UB}$*-*$\overline{LB}$ was also able to find the same classification rate and was 40% faster than *DF-*$\overline{UB}$*-*$\overline{LB}$.

From all the aforementioned results, one can conclude that even if the deviation of *DF* and *PDFS* was better when compared to *BP*, it did not have

	GREC		Protein	
	R	Time (ms)	R	Time (ms)
DF-$\overline{UB}$-$\overline{LB}$	0.98	171401.54	0.44	128469.57
DF-UB-$\overline{LB}$	0.98	163979.45	0.52	124361.61
DF-$\overline{UB}$-LB	0.98	140675.00	0.40	147371.86
DF-UB-LB	0.98	140525.48	0.52	145779.68
PDFS-UB-LB	**0.98**	**99850.79**	**0.52**	**80038.33**
A^*-$\overline{LB}$	0.89	358158.76	0.29	1065106.80
A^*	0.53	222045.94	0.26	194021.88
BS1	0.98	69236.34	0.24	129571.76
BS10	0.94	83928.21	0.26	139294.88
BS100	0.58	83928.20	0.26	141265.41
BP	**0.98**	**62294.60**	**0.52**	**59041.84**
H	0.96	63563.74	0.43	71990.62

Table 10: Classification on GREC and Protein. The best exact and approximate methods are marked in bold style. Note that the response time is the average time needed to classify each test graph

	MUTA	
	R	Time (ms)
DF-$\overline{UB}$-$\overline{LB}$	0.70089	1139134.29
PDFS-$\overline{UB}$-$\overline{LB}$	**0.70**	**760861.51**
A^*-$\overline{LB}$	0.4574	856793.020
BS-1	0.55	1015688.00
BS-10	0.55	1256793.02
BS-100	0.55	1383838.66
BP	**0.70**	**528546.64**
H	0.58	525610.25

Table 11: Classification on MUTA. The best exact and approximate methods are marked in bold style

an effect on the classification rate. In other words, for such an application, one does not need to have a very accurate algorithm in order to obtain a good classification rate.

6. Conclusion and Perspectives

In the present paper, we have considered the problem of GED computation for pattern recognition. GED is a powerful and flexible paradigm that has been used in different applications in PR. The exact algorithm A^*, presented in the literature suffers from high memory consumption and thus is too costly to match large graphs. In this paper, we propose a parallel exact GED algorithm, referred to as *PDFS*, which is considered as an extension of a recent GED method based on depth-first tree search (Abu-Aisheh et al., 2015). The algorithm in (Abu-Aisheh et al., 2015), referred to as *DF*, does not exhaust memory as the space complexity in the worst case is quadratic in the number of vertices i.e., $O(|V_1| \times |V_2|)$. In this paper, we speed up the computation of *DF* by adopting a load balancing strategy. Each thread gets one or more partial edit path and all threads solve their assigned edit paths in a fully parallel manner. A work stealing or balancing process is performed whenever a thread finishes all its assigned threads. Moreover, synchronization is applied in order to ensure upper bound coherence.

In the experiments part, we proposed to evaluate both exact and approximate GED approaches under large and small time constraints, on 4 publicly available datasets (GREC, MUTA, Protein and PAH). Such constraints are

devoted to accuracy and speed tests, respectively. Small time constraints ensured that the approximate methods *BP* and *H* were able to find a solution. Experiments demonstrated the importance of the load balancing strategy when compared to a naive method that does not include neither static nor dynamic load balancing. Under small and large time constraints, *PDFS* proved to have the minimum deviation, the maximum number of best found solutions and the maximum number of optimal solutions. However, since our goal was to elaborate methods dealing with rich and complex attributed graphs, *BS* was slightly superior to *PDFS* in terms of deviation when evaluated on the MUTA dataset under large time constraint. This could be improved by learning the best sorting strategy for a given database. Results 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 methods. On the other hand, *DF* and *PDFS* take longer time but lead to better results (except on MUTA). By limiting the run-time, our exact method provides (sub)optimal solutions and becomes an efficient upper bound approximation of GED with the same classification rate found by the best approximate method. Even though *DF* and so *PDFS* were more accurate than *PDFS*, their classification rate was as good as the best approximate GED (i.e., *BP*). On this basis, one could ask: What is the benefit of having a more precise algorithm in an a classification context?

A future promising work could be to make *PDFS* more scalable to have more precise and thus more optimal solutions under large time constraints.

This could be achieved by extending *PDFS* from a single machine algorithm to a multi-machines one. Moreover, learning to sort vertices of G_1 based on the structure and characteristics of graphs is another promising perspective towards a faster exact algorithm.

References

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

Abu-Aisheh, Z., Raveaux, R., Ramel, J.-Y., Martineau, P., 2015. An exact graph edit distance algorithm for solving pattern recognition problems, 271–278.

Allen, R., C. L. M. S. T. S. S. L., Yasuda, D., 1997. A parallel algorithm for graph matching and its maspar implementation. IEEE Transactions on Parallel and Distributed Systems 8 (5), 490–501.

Almohamad, H. A., Duffuaa, S. O., 1993. A linear programming approach for the weighted graph matching problem. IEEE Trans. Pattern Anal. Mach. Intell. 15 (5), 522–525.

Bertsekas, D. P., Tsitsiklis, J. N., 1997. Parallel and Distributed Computation: Numerical Methods. Athena Scientific.

Bougleux, S., Brun, L., Carletti, V., Foggia, P., Gauzerec, B., Vento, M., 2016. Graph edit distance as a quadratic assignment problem. Pattern Recognition Letters.

Boukedjar, A., Lalami, M., El-Baz, D., 2012. Parallel branch and bound on a cpu-gpu system. In: Parallel, Distributed and Network-Based Processing (PDP). pp. 392–398.

Brun, L., 2012. Relationships between graph edit distance and maximal common structural subgraph.

Bunke, H., 1997. On a relation between graph edit distance and maximum common subgraph. Pattern Recognition Letter. 18, 689–694.

Chakroun, I., Melab, N., 2013. Operator-level gpu-accelerated branch and bound algorithms. In: ICCS. Vol. 18.

Chung, C.-S., Flynn, J., Sang, J., 2012. Parallelization of a branch and bound algorithm on multicore systems. journal of Software Engineering and Applications 5, 12–18.

Cormen, T. H., et al., 2009. Introduction to Algorithms, 3rd Edition. The MIT Press.

Cortés, X., Serratosa, F., 2015. Learning graph-matching edit-costs based on the optimality of the oracle's node correspondences. Pattern Recognition Letters 56, 22–29.

Deng, L., Yu, D., 2014. Deep learning: Methods and applications. Found. Trends Signal Process. 7, 197–387.

Dorta, I., Leon, C., Rodriguez, C., 2003. A comparison between mpi and openmp branch-and-bound skeletons. In: Parallel and Distributed Processing Symposium, 2003. Proceedings. International. pp. 66–73.

Drozdowski, M., 2009. Scheduling for Parallel Processing, 1st Edition. Springer Publishing Company, Incorporated.

Fankhauser, S., Riesen, K., Bunke, H., Dickinson, P. J., 2012. Suboptimal graph isomorphism using bipartite matching. IJPRAI 26 (6).

Ferrer, M., Serratosa, F., Riesen, K., 2015. A first step towards exact graph edit distance using bipartite graph matching. In: Graph-Based Representations in Pattern Recognition - 10th IAPR-TC-15 International Workshop. pp. 77–86.

Fischer, A., Suen, C. Y., Frinken, V., Riesen, K., Bunke, H., 2013. A fast matching algorithm for graph-based handwriting recognition. Graph-Based Representations in Pattern Recognition, 194–203.

Fischer, A., Suen, C. Y., Frinken, V., Riesen, K., Bunke, H., 2015. Approximation of graph edit distance based on hausdorff matching. Pattern Recognition 48 (2), 331–343.

Gauzere, B., Brun, L., Villemin, D., 2012. Two new graphs kernels in chemoinformatics. Pattern Recognition Letters 33 (15), 2038 – 2047.

H. Bunke, G. A., 1983. Inexact graph matching for structural pattern recognition. Pattern Recognition Letters. 1, 245–253.

Justice D, H. A., 2006. A binary linear programming formulation of the graph edit distance. IEEE Trans Pattern Anal Mach Intell. 28, 1200–1214.

Kumar, V., Gopalakrishnan, P. S., Kanal, L. N. (Eds.), 1990. Parallel Algorithms for Machine Intelligence and Vision. Springer-Verlag New York, Inc., New York, NY, USA.

Leordeanu, M., Hebert , M., Sukthankar, R., 2009. An integer projected fixed point method for graph matching and map inference. In: Proceedings Neural Information Processing Systems. pp. 1114–1122.

Liu, Z., Qiao, H., 2014. GNCCP - graduated nonconvexityand concavity procedure. IEEE Trans. Pattern Anal. Mach. Intell. 36, 1258–1267.

M. Neuhaus, K. R., Bunke., H., 2006. Fast suboptimal algorithms for the computation of graph edit distance. Proceedings of 11th International Workshop on Structural and Syntactic Pattern Recognition. 28, 163–172.

Neary, M. O., Cappello, P. R., 2005. Advanced eager scheduling for java-based adaptive parallel computing. Concurrency - Practice and Experience 17, 797–819.

Neuhaus, M., Bunke., H., 2007. Bridging the gap between graph edit distance and kernel machines. Machine Perception and Artificial Intelligence. 68, 17–61.

Rao, V. N., Kumar, V., 1987. Parallel depth-first search on multiprocessors part i: Implementation. International journal on Parallel Programming 16(6), 479–499.

Riesen, K., B. H., 2008. Iam graph database repository for graph based pattern recognition and machine learning. Pattern Recognition Letters. 5342, 287–297.

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

Riesen, K., 2015. Structural Pattern Recognition with Graph Edit Distance - Approximation Algorithms and Applications. Advances in Computer Vision and Pattern Recognition. Springer.

Riesen, K., Bunke, H., 2014. Improving approximate graph edit distance by means of a greedy swap strategy. In: ICISP. pp. 314–321.

Riesen, K., Fankhauser, S., Bunke, H., 2007. Speeding up graph edit distance computation with a bipartite heuristic. In: MLG.
URL http://dblp.uni-trier.de/db/conf/mlg/mlg2007.html#RiesenFB07

Riesen, K., Fischer, A., Bunke, H., 2014. Improving approximate graph edit distance using genetic algorithms. In: SSSPR14. pp. 63–72.

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

Serratosa, F., 2015. Speeding up fast bipartite graph matching through a new cost matrix. IJPRAI 29 (2).

Tsai, W.-h., Member, S., Fu, K.-s., 1979. Pattern Deformational Model and Bayes Error-Correcting Recognition System. IEEE Transactions on Systems, Man, and Cybernetics 9, 745–756.

Van Loan, C., 1992. Computational Frameworks for the Fast Fourier Transform.

W. Christmas, J. K., Petrou., M., 1995. Structural matching in computer vision using probabilistic relaxation. IEEE Trans. PAMI, 2, 749–764.

Xu, C., Lau, F. C., 1997. Load Balancing in Parallel Computers: Theory and Practice. Kluwer Academic Publishers.

Zeng, Z., Tung, A. K. H., Wang, J., Feng, J., Zhou, L., 2009. Comparing stars: On approximating graph edit distance. Proc. VLDB Endow. 2, 25–36.