Efficient k-nearest neighbors search in graph space

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

Efficient k-nearest neighbors search in graph space

Zeina Abu-Aisheh Romain Raveaux
Jean-Yves Ramel
Laboratoire d'Informatique Fondamentale et Appliquées de Tours (EA 6300),
64 avenue Jean Portalis, Tours, France

May 14, 2018

email : {name.surname}@univ-tours.fr

Abstract

The k-nearest neighbors classifier has been widely used to classify graphs in pattern recognition. An unknown graph is classified by comparing it to all the graphs in the training set and then assigning it the class to which the majority of the nearest neighbors belong. When the size of the database is large, the search of k-nearest neighbors can be very time consuming. On this basis, researchers proposed optimization techniques to speed up the search for the nearest neighbors. However, to the best of our knowledge, all the existing works compared the unknown graph to each train graph separately and thus none of them considered finding the k nearest graphs from a query as a single problem. In this paper, we define a new problem called multi graph edit distance to which k-nearest neighbor belongs. As a first algorithm to solve this problem, we take advantage of a recent exact branch-and-bound graph edit distance approach in order to speed up the classification stage. We extend this algorithm by considering all the search spaces needed for the dissimilarity computation between the unknown and the training graphs as a single search space. Results showed that this approach drastically outperformed the original approach under limited time constraints. Moreover, the proposed approach outperformed fast graph edit distance algorithms in terms of average execution time especially when the number of graphs is tremendous.

1 Introduction

Graphs are frequently used in various fields of computer science, since they constitute a universal modeling tool for the description of structured data. The handled objects and their relations are described in a single and human-readable formalism. Hence, tools for supervised graphs classification and graph mining are required in many applications such as pattern recognition (Riesen, 2015a), chemical components analysis (Gaüzère et al., 2012) and structured data retrieval (Kooli and Belaïd (2016)).

Graph classification consists in assigning a class or a category to an unknown graph. To do so, the unknown graph is compared to the set of training graphs. The graph classification can be achieved directly in graph space ($\mathbb{G}$) or indirectly in vector space ($\mathbb{R}^n$) by means of an embedding method. Methods operating in a vector space can also be split into two parts whether they rely on explicit ($\phi : \mathbb{G} \rightarrow \mathbb{R}^n$) (Gaüzère et al., 2012) or implicit ($k :< \mathbb{G}, \mathbb{G} >\rightarrow \mathbb{R}$) graph embedding (Riesen et al., 2010). Concerning the explicit graph embedding methods, some features (vertex degree, labels occurrence histograms,etc.) are extracted from the graph. Hence, the graph is projected in a Euclidean space. The choice of sufficient features is not trivial. Moreover, the number of such features has to be very large and thus the dimensionality issues occur. Regarding the implicit approaches, an explicit data representation is of secondary interest. That is, rather than defining individual representations for each graph, the graph at hand is represented by pairwise comparisons only. The graphs are implicitly projected in a Euclidean space without defining the function ϕ. However, such approaches suffer from their computationally intensive cost when the dataset is large. This paper deals with paradigms that operate directly on the graph space and can thus capture more structural distortions.

One of the most well-known and used approaches to compute a distance (dissimilarity) between two graphs taking distortion into account is the graph edit distance (GED). The GED is achieved by finding a set of graph edit operations: insertions, deletions and substitutions of vertices as well as edges in order to transform a graph into another with the minimal cost. The traditional approach to graph edit distance-based pattern recognition system is given by the k-Nearest Neighbor classification (kNN) (Riesen (2015a)). This classifier is simple in the sense that only a metric, or a distance function, that measures the dissimilarity between graphs has to be defined. Afterwards, the unknown graph is compared to all the graphs in the training set. Finally, the unknown graph is assigned to the most common class among its kNN measured by the distance function.

Finding the exact kNN can be done through the calculation of an exact GED (such as in Abu-Aisheh et al. (2015); Neuhaus et al. (2006a)). This approach has an exponential complexity in function of the size of compared graphs and a linear complexity in function of the number of training graphs. Moreover, its complexity increases especially when the number of graphs in the learning set is tremendous. Two approaches could be conducted to accelerate the process of the kNN. One can cluster and structure the search space (e.g., Wang (2012)) or use graph prototypes so as to avoid comparing a test graph with all the graphs in the training set (e.g., Raveaux et al. (2011)). Another approach could be to find a fast GED algorithm. Recently, lots of approximate GED methods have been proposed in the literature (Riesen (2009); Fischer et al. (2015); Bougleux et al. (2017)) to cite a few of them. However, in these approaches all the comparisons are performed independently regardless to the final kNN objective and the obtained solutions are not exact. In this paper, we propose to reformulate the kNN problem under the multi graph edit distance (MGED) paradigm. During the classification of a test graph, all the GED computations whose objective is to classify an unknown graph G_i are merged to obtain a more global problem. As a first algorithm to solve the MGED problem, we take advantage of a recent exact branch-and-bound (BnB) GED approach, called DF in (Abu-Aisheh et al., 2015). The knowledge about the global MGED problem can be considered as the merging of the search spaces to prune the global search space associated to all sub-GED problems.

This paper is organized as follows: In Section 2, the kNN classifier and the GED problem are presented. In Section 3, we spot light on the fast GED metrics approaches in the literature that are currently used in a classification context. In Section 4, our proposal is detailed. First, a formal definition of the MGED problem is given. Second, a first algorithm to solve the MGED problem is put forward. This method is an extension of an existing BnB GED strategy. The purpose of the new algorithm is to speed up the kNN search in graph space. The parameters' impact of the algorithm is then discussed. Section 5 is dedicated to the experiments, protocol and results that show the efficiency of the approach in terms of accuracy and computation time. Section 6 is devoted to conclusions and perspectives.

2 Problem statement

Our proposal lies in accelerating the classification process of the kNN classifier in graph space. Thus, in this section, we formally define each of the kNN classifier (Bhatia and Vandana (2010)) and the GED problem (Riesen (2015a)).

2.1 The k-nearest neighbors problem

The objective of the kNN classifier is to classify an unknown object by assigning it a class which represents the majority of its nearest neighbors. Let the objects be graphs and let $\mathcal{D}$ be the set of graphs and let $\mathcal{C}$ be the set of classes. Given a graph training set $TrS = \{(G_j, c_j)\}_{j=1}^{M}$, where $G_j \in \mathcal{D}$ is a graph and $c_j \in \mathcal{C}$ is the class of the graph. The kNN classifier induces from TrS a mapping function $f : \mathbb{G} \to \mathcal{C}$ which assigns a class to an unknown graph from the test set TeS. The 1-nearest neighbor problem can be defined as follows:

Definition 1 *1-Nearest Neighbors Problem*

$$(G^*, c^*) = arg \min_{(G_j, c_j) \in TrS} d(G_i, G_j) \;\; \forall G_i \in TeS \tag{1}$$

Where $d(G_i, G_j)$ is the metric used to calculate a dissimilarity between G_i and G_j. In Definition 1, the number of calls to the dissimilarity function is equal to M, where M is the size of the training set. To extend Definition 1 to k-nearest neighbors, we introduce $\mathcal{K}$, the set of the kNN from a query graph $G_i \in TeS$. Let $\mathcal{K} = \{(G_1, c_1), \cdots, (G_j, c_j), \cdots, (G_k, c_k)\}$ be a set of graphs along with their class labels with $(G_j, c_j) \in TrS$. The k-Nearest Neighbors problem can be defined by:

Definition 2 *k-Nearest Neighbors Problem*

$$\mathcal{K} = arg \underset{(G_j,c_j)\in TrS}{sort}(d(G_i,G_j),k) \;\; \forall G_i \in TeS \tag{2}$$

Where *sort* is a function that performs an ascending sort of $d(G_i, G_j)$ values. k is the number of retained values to choose the number of nearest neighbors of G_i. To exploit the Definition 2 in a classification context, a voting operator has to be defined. The max voting operator is a function $\tau : \mathcal{K} \rightarrow \mathcal{C}$ defined by:

Definition 3 *Max Voting Operator* $:\tau$

$$c^*_j = arg \max_{c_j\in\mathcal{C}} m_{c_j}(\mathcal{K}) \tag{3}$$

Where m_{c_j} is a function that counts the number of observations that fall into each class c_j.

The kNN classifier has been widely used in the literature. The use of this classifier is very simple since it is a non-parametric technique and thus it does not need knowledge about the distribution of classes. Moreover, when the metric is defined, the kNN classifier can provide an explanation of the classification results and thus the kNN classifier has an advantage over the other classifiers which are considered as black-box models (Dreiseitl and Ohno-Machado (2002)). In (Cover and Hart, 2006), it has been shown that when there are enough training patterns, the classification error of the kNN classifier is smaller than twice the Bayes error.

2.2 The graph edit distance problem

In graph space, the similarity or dissimilarity between two graphs requires the computation and the evaluation of the *best* matching between them. Since exact isomorphism rarely occurs in pattern analysis applications, the matching process must be error-tolerant, i.e., it must tolerate differences on the topology and/or its labeling. In this context, the most well-known paradigm in the literature is the graph edit distance (GED) (Riesen, 2015a). In the GED, the graph matching process and the dissimilarity computation are linked through the introduction of a set of graph edit operations. Each edit operation (i.e., substitution, deletion and insertion of vertices and edges) is characterized by a cost, and the dissimilarity measure is the total cost of the least expensive set of operations that transform one graph into another one.

Formally saying, the GED between two attributed graphs G_i and G_j is defined as follows:

Definition 4 *Graph Edit Distance*

$$d_{\lambda_{min}}(G_i, G_j) = \min_{\lambda\in\Gamma(G_i,G_j)} \sum_{o\in\lambda} c(o) \tag{4}$$

where $c(o)$ denotes the cost function measuring the strength of an edit operation o and $\Gamma(G_i, G_j)$ denotes the set of all the edit paths transforming G_i into G_j. The exact correspondence is one of the correspondences that obtains the minimum cost (i.e., $d_{\lambda_{min}}(G_i, G_j)$). The GED problem formulated in Definition 4 has been proven to be NP-Hard in (Zeng et al., 2009). As mentioned in Section 2.1, when the number of graphs in the training set is large, the kNN classifier becomes a time-consuming process. In fact, the number of calls to the GED solver grows linearly in function of the training set size. Moreover, the GED problem cannot be solved optimally in polynomial time. Consequently, many researchers have focused their research on designing fast KNN classifiers on top of fast GED solvers. Thus, in the next section, we explore the relatively fast GED methods proposed as a dissimilarity measure that are usually used when classifying graphs using the kNN classifier.

3 State of the art: GED approaches as a dissimilarity measure to classify graphs using kNN neighbors

To propose fast GED algorithms that help in speeding up the classification process, researchers switched from exact to approximate GED methods.

To overcome the bottleneck of the best first BnB algorithm referred to as A^* in (Neuhaus et al., 2006a), a recent GED algorithm, referred to as *DF*, was put forward in (Abu-Aisheh et al., 2015) to reduce the memory consumption and also the computation time. This was done using a different exploration strategy (i.e., depth-first instead of best-first). Furthermore, the unfruitful nodes in the search tree are pruned by a lower and upper bounds strategy. This algorithm was transformed to anytime *DF* in (Abu-Aisheh et al., 2016) thanks to the time constraints where the user can stop the algorithm at a specific t and output the encountered solutions.

Beam-Search (*BS*) in (Riesen (2009)) has been put forward to reduce the complexity of A^*. The purpose of BS is to prune the search tree via a parameter that keeps the x most promising partial edit paths. The assignment problem in (Riesen, 2009) was reformulated as finding an exact matching in a complete bipartite graph in order to reduce the quadratic assignment problem (of GED computation) to an instance of a linear sum assignment problem. This method was then sped up in (Serratosa (2014, 2015)). In the fast *BP* in (Serratosa (2014)), referred to as *FBP*, the cost matrix is composed of only one quadrant whereas in (Serratosa (2015)) two square matrices are defined depending on the order of the involved graphs. These approaches take local rather than global relationships into consideration. To go beyond the local structure problem, few works have been proposed (Riesen et al. (2014); Ferrer et al. (2015); Carletti et al. (2015)), to name a few of them. However, the computation time of these approaches is higher than *BP* and *FBP*. Recently, two approaches based on Integer Projected Fixed Point and Graduated Non-Convexity and Concavity methods have been proposed in (Bougleux et al., 2017).

All these approaches are relatively fast when we intend to compare a single graph pair. However, in a classification context, especially when the number of graphs is large, the sum of the number of calls to the GED solvers leads to an expensive computational cost. To the best of our knowledge, none of the previous works dealt with the problem of classifying a test graph as a single and global graph comparison problem. In a preliminary work in Abu-Aisheh et al. (2017), a new problem referred to as multiple GED (MGED) was defined. This approach was limited to $k = 1$. In this paper, we extend this problem to kMGED by generalizing the problem to $k \geq 1$. We then propose a first algorithm to solve the kMGED problem. Finally, in this paper a stronger experimental study is provided.

4 Our proposal: Solving the kMGED problem

When using the kNN classifier to classify graphs, the comparison of G_i and each G_j is achieved independently. That is, the result of a prior comparison cannot help in solving the next comparison. To tackle this problem, we propose to define the MGED problem along with a dedicated algorithm.

4.1 A new problem: Reformulating the kNN search as the kMGED problem

The MGED can be seen as merging of the two problems formulated in Definitions 2 and 4. First, let us recall that $\Gamma(G_i, G) = \{\lambda^1_{G_i,G}, \lambda^2_{G_i,G}, \cdots, \lambda^p_{G_i,G}\}$ is the set of all possible matchings between G_i and G. The number of possible matchings p is exponential with respect to the number of vertices in G_i and G. Now, let $\mathcal{L}_i = \{(c_1, \Gamma(G_i, G_1)), \cdots, (c_j, \Gamma(G_i, G_j)), \cdots, (c_M, \Gamma(G_i, G_M))\}$ be the set of all possible matchings between G_i and each graph $G_j \in TrS$ where c_* is the class of the graph G_*. The set $\mathcal{L}_i$ can be expanded by developing the Γ sets. $\mathcal{L}_i = \{(c_1, \lambda^1_{G_i,G_1}), (c_1, \lambda^2_{G_i,G_1}), \cdots, (c_1, \lambda^p_{G_i,G_1}), \cdots, c_j, \lambda^1_{G_i,G_j}),$ $(c_j, \lambda^2_{G_i,G_j}), \cdots, (c_j, \lambda^q_{G_i,G_j}), \cdots, (c_M, \lambda^1_{G_i,G_M}), (c_M, \lambda^2_{G_i,G_M}), \cdots,$
$(c_M, \lambda^r_{G_i,G_M})\}$. The MGED problem can be defined as follows:

Definition 5 *Multi Graph Edit Distance Problem*

$$(G_*, c_*) = arg \min_{(c,\lambda_{G_i,G}) \in \mathcal{L}} \sum_{o \in \lambda_{G_i,G}} c(o) \;\; \forall G_i \in TeS \tag{5}$$

The former Definition 5 can be seen as searching the minimum matching among all the matchings between one query graph G_i and a graph collection. In Definition 2, the MGED problem can be extended to kNN by using the *sort* function. The kMGED problem can be defined as follows:

Definition 6 *k-Multi Graph Edit Distance Problem*

$$\mathcal{K} = arg \underset{\lambda_{G_i,G}\in\mathcal{L}}{sort}\Big(\sum_{o\in\lambda_{G_i,G}} c(o), k\Big) \;\; \forall G_i \in TeS \tag{6}$$

where *sort* is a function that performs an ascending sort. From the list of elements outputted by the *arg sort* function, only the k best elements are conserved with respect to the following uniqueness quantification constraint:

$$\exists!(G_j, c_j) \in \mathcal{K} \tag{7}$$

where $\mathcal{K}$ is the set of graphs along with their class labels with $(G_j, c_j) \in TrS$. Constraint 7 ensures that the pair (G_j, c_j) cannot appear twice. That is, only the best feasible solutions of each GED computation (G_i and G_j) is selected and thus (G_j, c_j) appears only once. In the worst case, the time complexity of solving the problem of kMGED is exponential in the number of vertices of the graphs G_i and G_j multiplied by the number of graphs in TrS. In other words, the complexity in the worst case equals to the complexity of the kNN classifier (see Definition 2) multiplied by the complexity of the GED problem (see Definition 4).

4.2 Our proposal: One-tree depth first algorithm to solve the kMGED problem

In this section, we put forward a first algorithm to solve the kMGED problem defined in Definition 6. As a final application, the algorithm uses the kNN to classify a query graph G_i with a single search space.

4.2.1 Algorithm description

Algorithm 1 depicts the main steps of the proposed algorithm, called *One-Tree-kMGED*. Lines 1 to 3 correspond to the initialization step. The GED solver is called for the G_i and G_j (Line 5). The obtained distance d is then added to the list $Dmin$ at the location $k+1$ (line 7). The list of distances is sorted in ascending order while keeping track of IDs (Line8). In Line 9, the upper bound UB is updated and is given the value of the k^{th} element saved in the distance list $Dmin$ (i.e., $Dmin[k]$). After all the aforementioned steps, the algorithm *One-Tree-kMGED* returns the graphs along with their associated class label $(G_{IDmin[k]}, c_{IDmin[k]})$.

Algorithm 1 One-Tree-kMGED Algorithm

Input: The set TrS: $\{(G_1, c_1), \cdots, (G_M, c_M)\}$, the unknown graph G_i and the parameter k
Output: the k nearest graphs to G_i from the set TrS with their associated class

```
 1: $Dmin = [+\infty, \cdots, +\infty]$                          ▷ A distance of $k+1$ elements
 2: $IDmin = [+\infty, \cdots, +\infty]$                                    ▷ A graph ID list
 3: $UB = +\infty$                                                ▷ The initial upper bound
 4: for $j = 1$ to $M$ do
 5:     $d$ = GED ($G_i$,$G_j$,$UB$)
 6:     $Dmin[k+1] = d$
 7:     $IDmin[k+1] = j$
 8:     $(IDmin, Dmin) = sort_{Dmin}(IDmin, Dmin)$              ▷ sort in an ascending order
 9:     $UB = Dmin[k]$
10: end for
11: Return $(G_{IDmin[1]}, c_{IDmin[1]})$, $\cdots$, $(G_{IDmin[k]}, c_{IDmin[k]})$
```

4.2.2 Used GED solver

In line 5 of Algorithm 1, any GED solver that takes an upper bound as an input is suitable. We propose to use the depth first algorithm *DF* in (Abu-Aisheh et al., 2015) since this algorithm outperformed A^* in terms of time and memory consumption in addition to its ability to prune the search space thanks to its upper and lower bounds. *DF* consists of two main steps: Preprocessing and branch-and-bound. The preprocessing step consists in speeding up the calculations through the construction of the vertex-to-vertex assignment cost matrix (the same applies on edges). Once the preprocessing stage is finished. The solution space is organized as a search tree. The exploration of the search tree is performed in a depth-first way. Bounding is performed when finding a leaf node with a cost smaller than UB. Pruning is performed by cutting nodes with a larger cost than UB. A node p has a cost computed as follows: $lb(p) = g(p) + h(p)$ where $g(p)$ is the cost of the partial edit path and $h(p)$ is an estimation of the future cost to obtain a complete edit path. If $lb(p)$ is not a lower bound, DF can miss optimal solutions. $lb(p)$ is a lower bound if it does not overestimate the optimal cost of a complete edit path. Consequently, $lb(p)$ is a lower bound if $h(p)$ does not overestimate the remaining cost of unmatched elements. $h(p)$ is computed as follows: Let us assume that a partial edit path at a position in the search tree is given, and let the number of unprocessed (unmatched) vertices of the first graph G_i and second graph G_j be n_i and n_j, respectively. To estimate the costs of the remaining optimal edit operations, we accumulate the costs of the $min(n_i, n_j)$ least expensive node substitutions. The linear assignment problem should be solved to achieve this goal. However, to be fast, the minimization problem is not solved. Instead, we consider that each substitution is cost-free. Consequently, the least expensive node substitutions are always equal to zero. Any of the selected substitutions is always cheaper than a deletion or an insertion operation. Next, we accumulate the costs of $max(0, n_i - n_j)$ node deletions and $max(0, n_j - n_i)$ node insertions. The unprocessed edges of both graphs are handled analogously and independently of the vertices. 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_i into the remaining part of G_j. But, the estimated cost certainly constitutes a lower bound of the exact cost. This fast lower bound is used to make the algorithm as fast as possible and to spend the available time exploring and pruning the global search tree rather than computing the lower bound.

In order to simply illustrate *One-Tree-kMGED*, Figure 1 highlights its idea for $k = 1$. Given a query graph G_i and a learning database TrS, the idea is to consider each search tree S_{ij} of the $d(G_i,G_j)$ as a sub-tree of the global tree dedicated to G_i and referred to as T_{ij}. For instance, in Figure 1, one can see that the first UB found while exploring the sub-tree S_{i1} of GED(G_i,G_1) is 2. UB is then used as an initial UB of the sub-tree S_{i2} of GED(G_i,G_2) and so on. Such an operation helps in pruning the sub-trees as fast as possible while searching for the nearest neighbor of G_i.

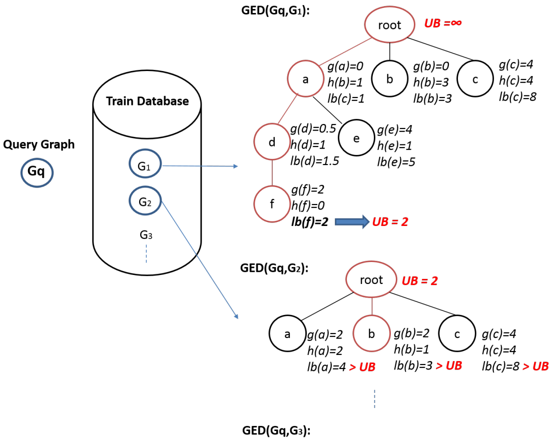

Figure 1: *one-Tree-kMGED* when $k = 1$. Given a query graph G_i and graphs in the training set, the problems GED(G_i,G_1), GED(G_i,G_2) and GED(G_i,G_3) are considered as sub-trees of the global tree (T_{G_i}). The sub-tree of GED(G_i,G_j) is pruned thanks to UB that is found via GED(G_i,G_1).

4.2.3 Complexity and time constraint

In the worst case, the time complexity of *one-Tree-kMGED* is exponential in the number of vertices of the involved graphs. This case occurs when the first UB does not help in pruning the rest of the search tree and thus all the TrS subtrees need to be explored. One should notice that the complexity in the worst case equals to the complexity of the naive way illustrated in Definition 2.

Conceptually speaking, DF and *One-Tree-kMGED* provide the same neighbors. This statement is true under one assumption that is the time limited to solve each graph comparison is infinite. Another way to view this assumption is to say that the time limit is never reached by the solvers. On the other hand, *One-Tree-kMGED* does not output the distance of each graph pair (G_q,G_i). The GED computation of each (G_q,G_i) is stopped as soon as the solver proved that no better solutions than the global upper bound UB could be found. In such a case, the UB value is returned as an output of the given graph comparison. The only distances that are guaranteed to be outputted are the GED values of the kNN. The global UB could accelerate the classification time and improve the classification rate.

4.3 Theoretical discussion around the parameters impacts

As depicted in Algorithm 1, *one-Tree-kMGED* has 3 parameters: the unknown graph G_i, the training set *TrS* and the parameter k for the selection of the nearest neighbors. In this section, a discussion about the impact of *Trs* and k is provided

4.3.1 Parameter k

The parameter k has an important impact on the classification rate of kNN. This impact was deeply studied in (Batista and Silva (2009)). The authors showed that on 4 different types of datasets the classification rate increases as k increases up to a maximum between k=5 and k=11. Suppose we have a classifier with only 2 classes, using less neighbors (e.g., k=1) could lead to overfitting. This phenomena will be also demonstrated in the experiments.

Regarding *One-Tree-kMGED*, having a big value of k, could have a big impact not only on the classification rate but also on the execution time. The reason is that, the upper bound will be the kth one in the $\{d_{min}\}$ list and not the best upper bound found so far. The kth upper bound is higher than the first upper bound. Consequently, the kth upper bound is likely less capable of cutting the search tree. Such a fact could slow down the algorithm depending on the difficulty of the classification problems.

4.3.2 Ordering the graphs in the training set

In Algorithm 1, the graphs in *TrS* were supposed to be already ordered. However, the question arises: Which order of training graphs should be taken into account in order to prune the search tree as soon as possible?

In this article, we propose 3 different orderings. Two of them depend on the class c_j of each graph G_j in *TrS*. Formally saying:

$$\begin{aligned} TrS_{CACO} = \{(G_n, c_n)\}_{n=1}^{M} = \quad & sort_{c_j}(TrS = \{(G_j, c_j)\}_{j=1}^{M}) \\ s.t. \quad & c_n < c_{n+1} \; \forall n \in [1, \cdots, M-1] \end{aligned} \tag{8}$$

$$\begin{aligned} TrS_{SGPCO} = \{(G_n, c_n)\}_{n=1}^{M} = & \, mod_{c_j}(TrS = \{(G_j, c_j)\}_{j=1}^{M}) \\ & s.t. c_n = c_j \; mod(card(\mathcal{C})) \; \forall j \in [1, \cdots, M] \end{aligned} \tag{9}$$

$$TrS_{RO} = \{(G_n, c_n)\}_{n=1}^{M} = random(TrS = \{(G_j, c_j)\}_{j=1}^{M}) \tag{10}$$

where TrS_{****} is TrS with a different order. Let us suppose that we have 3 classes, $\mathcal{C} = \{1, 2, 3\}$. The first ordering technique we propose is called Class-After-Class-Order (CACO) reordering (see Eq.8). In this ordering technique, all the graphs whose class is 1 are put first in TrS_{CACO}, then the graphs whose class is 2 and so on. This approach might not lead to a fruitful pruning because if an unknown graph belongs to Class N which is put at the end of TrS_{CACO}, then its search tree would have so many nodes, the upper bound would be relatively big and the execution time would be long.

The second technique based on classes is called Single-Graph-Per-Class-Order (SGPCO) (see Eq.9). The Single-Graph-Per-Class assures to insert a graph whose class is 1 in TrS_{SGPCO} first ($c_1 = 1$), then a graph whose class is 2 ($c_2 = 2$), then a graph from 3 ($c_3 = 3$), then a graph from class 1 ($c_4 = 1$) and so on. Compared to the first reordering approach, this one is ideal in the sense that the distribution of graphs is homogeneous.

Another technique that is not based on classes is called Random-Order (RO) (see Eq.10). The idea is to randomly reorder the graphs in *TrS*, this approach is better than Class-After-Class-Order, however, it may not efficiently prune the search tree as Single-Graph-Per-Class because for instance a random reordering could put 3 graphs successively whose class is the same.

In the experiments section, a study around the reordering of graphs in *TrS* is conducted.

5 Protocol and experiments

This section is dedicated to the protocol and experiments that show the interest of *One-Tree-kMGED* in a graph classification context.

5.1 Selected datasets

In the experiments, 6 datasets (GREC, Protein, Mutagenicity, Fingerprint and Webpage) from the IAM repository (Riesen, 2008) and House-Hotel from the Tarragona repository (Moreno-García et al., 2016) are selected. GREC contains graphs of rather small size with continuous attributes on vertices and edges which play an important role in the matching process. MUTA is representative of graph matching problems where graphs have only symbolic attributed and the number of the train graphs is 1500 which is the largest in IAM. Protein contains numeric attributes on each vertex as well as a string sequence that is used to represent the amino acid sequence. Webpage has the biggest graphs in IAM where the maximum number of vertices is 785. Webpage has numeric values on both vertices and edges. Fingerprint has only attributes on edges. Finally, House-Hotel has 60-size feature vector using Context Shape on vertices and the distance between two points on edges.

Table 1 synthesizes the characteristics of each of the selected datasets in terms of the number of graphs in both train and test sets, the average and maximum number of vertices and edges and the attributes on both of them.

Table 1: The characteristics of the datasets included in the experiments.

Dataset	**GREC**	**Protein**	**Muta**	**Fingerprint**	**Webpage**	**House-Hotel**
#train graphs	286	200	1500	378	780	71
#test graphs	528	200	2337	1532	780	70
$\overline{vertices}$	11.5	32.6	30.3	5.38	186.04	30
$\overline{edges}$	12.2	30.8	79	8.8	104.03	62.1
Max vertices	25	40	71	26	785	30
Max edges	30	149	112	48	524	79
Vertex labels	x,y coordinates	Type and amino acid sequences	Chemical symbol	None	Word's frequency	60 size feature
Edge labels	Line type	Type and length	Valence	Orientation	Section label	Distance

Each dataset has specific edit cost functions that define how the insertion, deletion and substitution are achieved (Riesen, 2015b). In most of the datasets, two non-negative meta parameters are associated: (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 the edit operation cost on the vertices or on the edges is more important. Table 2 demonstrates the cost functions of each of the included datasets as well as their meta parameters.

Table 2: The cost functions and meta parameters of the datasets.

Dataset	**GREC**	**Muta**	**Protein**	**Fingerprint**	**Webpage**	**House-Hotel**
τ_{vertex}	90	11	11	0.7	2	3
τ_{edge}	15	1.1	1	0.5	2	3
α	0.5	0.25	0.75	0.75	0.5	0.5
Vertex substitution function	Extended euclidean distance	Dirac function	Extended string edit distance	Absolute value	Dirac function	Dirac function
Edge substitution function	Dirac function	Dirac function	Dirac function	Absolute value	Absolute Value	Dirac function
Reference of cost functions	Riesen (2009)	Riesen (2009)	Riesen (2009)	Riesen (2009)	Riesen (2009)	Moreno-García et al. (2016)

5.2 Chosen graph matching methods (coupled with classical KNN)

To compare our global approach with the classical way to use kNN with graphs, we select different exact and approximate GED algorithms.

On the exact method side, we chose the depth-first GED algorithm from (Abu-Aisheh et al., 2015) since it outperforms the A^* algorithm (Riesen et al., 2007) in terms of the running time and memory consumption.

On the approximate side, we included the beam-search (*BS*) algorithm with two different values *BS-1* (i.e., the greedy algorithm) and *BS-100*. We also chose the bipartite matching algorithm (*BP*) (Riesen, 2009) as it has been shown to be one of the most efficient approximate algorithms so far. In addition, we selected a fast version of *BP* (Serratosa, 2014), referred to as *FBP* in the literature. In our implemented version of *FBP*, the three restrictions on the edit costs were not included (Serratosa, 2015). Finally, we picked a set-based approach based on the Hausdorff matching (Fischer et al., 2015). Unlike other methods, this algorithm is unable to output a matching. Table 3 summarizes the chosen methods.

The lower bound of each of the branch-and-bound methods included in the experiments (i.e., *DF*, *BS-1* and *BS-100*) is the same one used in *one-Tree-kMGED*, see Section 4.2.2.

Table 3: Methods included in the experiments.

Acronym	Reference	Details
one-Tree-kMGED	This paper	First algorithm to solve the kMGED
DF	Abu-Aisheh et al. (2015)	Depth-First GED algorithm
BS-1 and *BS-100*	Neuhaus et al. (2006b)	Beam-search with the size of the open path stack limited to 1 and 100, respectively
BP	Riesen (2009)	The bipartite GM
FBP	Serratosa (2015)	Fast *BP*
H	Fischer et al. (2015)	A lower bound algorithm based on the Hausdorff distance

5.3 Environment and Constraints

The experiments were conducted on a computer with a 24-core Intel i5 processor at 2.10GHz and 16 GB of memory. The time constraint used for all the datasets is fixed to 500 milliseconds (ms) by dissimilarity computation that is the maximum time needed by *BP* and *FBP* to output a solution. This way ensures that *BP* and *FBP* could solve any instance. On this basis, any GED algorithm that needs more than 500 ms is stopped and the best answer found so far is outputted. Note that *One-Tree-kMGED* and *DF* are exact algorithms without time constraints.

In order to study the influence of the training graphs order on *One-Tree-kMGED*, the 3 different reordering ways discussed in Section 4.3.2 are integrated. Note that for the Class-After-Class-Order (CACO), only one reordering way is integrated. This reordering is the same as the one in the initial files of the IAM repository (Riesen and Bunke, 2009).

As for the single-graph-per-class ordering (SGPCO) where the order of graphs is homogeneous, the order of classes is selected according to the order of the classes given in the initial files of the IAM repository. Thus, the first graph given in the initial file is chosen first, then the first graph of the second class and so on.

Concerning the random ordering (RO), 4 different random orders are generated because we are interested in testing the impact of different random orders on *One-Tree-kMGED*.

5.4 Results

This section is 4-fold. First *One-Tree-kMGED* is compared to the classical approaches cited in Table 3. Second, the evolution of the upper bound and the average time per interval is tracked. Third, the study of the reordering techniques mentioned in Section 4.3.2 is conducted. Last but not least, the impact of varying the number of graphs in the training set and the parameter k is depicted.

5.4.1 Comparison with the classical approaches

In Table 4, the results achieved on all the datasets are presented. Note that the computation time corresponds to the average time needed per dissimilarity computation in milliseconds (ms). Moreover, in this experiment, k was fixed to 1.

The results show that on all the datasets, *one-Tree-kMGED* was always faster than the classical *DF* approach. It also improved the classification rate of *DF* on both Protein and Muta. *one-Tree-kMGED* could improve *UB* while moving from one comparison to another. As a consequence, it pruned unfruitful parts of the global search tree and found smaller distances. A better minimization of the GED or the MGED problems does not always lead to a higher classification rate but it is more probable to find significant neighbors. When comparing *one-Tree-kMGED* to *BP*, one can see that *one-Tree-kMGED* was 4.2 times faster (on GREC), 3.6 times faster (on Muta) and 9.5 times faster (on WebPage). On the other hand, on Fingerprint and House-Hotel, the speed results of the two methods were quite similar. This is due to the small number of graphs in these datasets and thus the advantage of using prior UB in *one-Tree-kMGED* cannot be fully revealed. Moreover, on House-Hotel, *BS-1* was the fastest. House-Hotel has easy graphs to classify (with only 2 classes) and that is why all the methods obtained 98.5% as a classification rate. Another interesting remark is that *one-Tree-kMGED* succeeded in improving the classification rate on Muta. However, on Protein, it was less accurate than *BP*. This is due to the number of train graphs which is 1500 on Muta and 200 on Protein as depicted in Table 1. On MUTA, *one-Tree-kMGED* was able to better minimize the sum of the costs of edit operations than *BP*. Despite the fact that *BS-1* was faster than *one-Tree-kMGED* on Protein, the accuracy of *BS-1* was lower. As a general conclusion, *one-Tree-kMGED* was the fastest algorithm, except on Protein and CMU where *FBP* won. This point is explained by the fact that the number of train graphs in both of them was quite small so that the interest of merging all sub-problems into a unique one is not useful, see Table 1. Thus, the search tree of *one-Tree-kMGED* corresponding to each G_i is relatively small when compared to the trees of the other datasets. *one-Tree-kMGED* has less chance/time to prune off the search tree and thus fast approximate methods like *FBP* could be faster. One could also see that the speed gap between *one-Tree-kMGED* and approximate methods such as *BP* and *FBP* increases when the size of graphs and the number of train graphs increases, as it is demonstrated on Muta and WebPage.

5.5 The evolution of the upper bound

We deeply studied what happens during the classification of test graphs using *one-Tree-kMGED*. Thus, first, we took three different graph queries (that were part of Table 1) from GREC, Muta and Fingerrint, respectively. Figure 5 depicts the upper bound improvement at the end of each comparison $DF(G_i, G_j)$ where $G_j \in TrS$ on the 3 datasets (see line 5 in Algorithm 1). The settings are as follows: the order is RO and $k = 1$. One can see that improving *UB* happens frequently in the first few milliseconds, however, after a certain time, *UB* stays longer time without being changed or with a small gap between the current *UB* and the new one.

5.5.1 The evolution of the average execution time

For the same classification problems illustrated in Figure 5 and under the same settings, we analyze the evolution (during different intervals) of the time needed to compare each pair of graphs while exploring the search tree. Figure 9 illustrates the results on GREC, Muta and Fingerprint, respectively. We can see that the first distance computation on GREC and Muta needed approximately 400 ms while the other comparisons needed less time. This shows that the *UB*s were fruitful in pruning the search tree. We could also notice that the average execution time of the intermediate comparisons on Muta is higher than the one on GREC. The reason for that is related to the pruning of the search tree of Muta which is not as easy as GREC. On the other hand, on Fingerprint, a different behavior is observed. In fact, as illustrated, the first comparison needed less time compared to the other successive comparisons during the exploration of the search tree. When digging into the details of the results, we noticed that on Fingerprint in general, the found upper bounds are relatively small compared to other datasets in the paper. Thus for such a small *UB*, sometimes it could prune the search tree from the beginning and sometimes not. Moreover, the gap between successive upper bounds is not big on Fingerprint. For instance, in our chosen

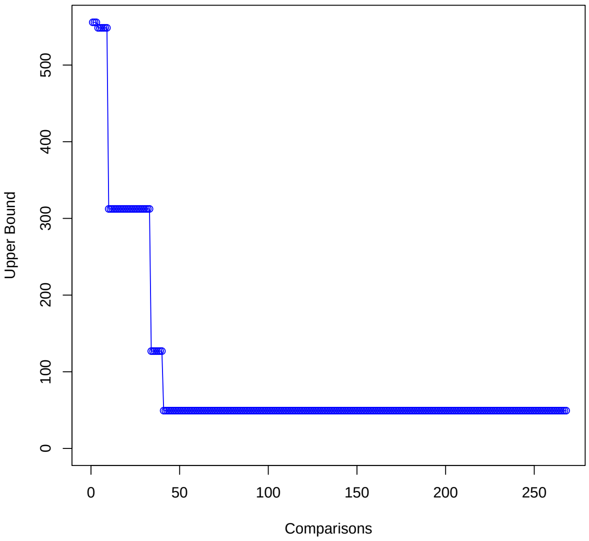

Figure 2: GREC

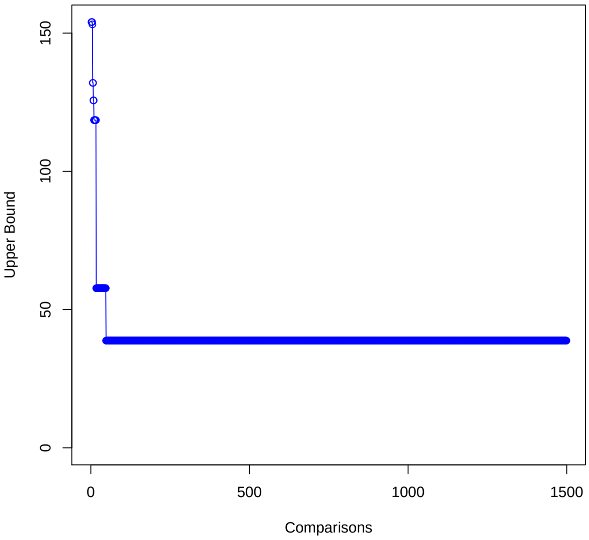

Figure 3: Muta

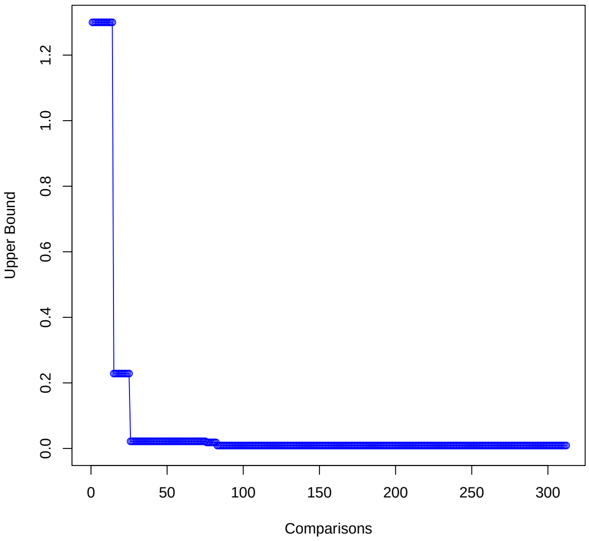

Figure 4: Fingerprint

Figure 5: *One-Tree-kMGED(RO)*: The upper bound found at the end of each comparison $DF(G_i,G_j)$.

G_i from Fingerprint and Muta, the gap between the first UB and the second one in Muta is 20.56 times more than the gap in Fingerprint.

5.5.2 The impact of reordering the train graphs

Regarding the RO, each train set, depicted in Section 5.1, was randomized 4 times. The results of these orders were quite similar. Thus, the measured time, shown in Table 4, is the average classification time of the 4 different orders. Concerning the different orders: MGPGO, SGPGO and RO, on all the datasets, *One-Tree-kMGED(SGPGO)* was always faster than *One-Tree-kMGED(MGPGO)*. On the other hand, regarding *One-Tree-kMGED(SGPGO)* and *One-Tree-kMGED(RO)*, no one beats the other. Theoretically saying and as discussed in Section 4.3, *One-Tree-kMGED(SGPGO)* should be faster than *One-Tree-kMGED(RO)* especially when the number of training graphs is high. In practice, *One-Tree-kMGED(SGPGO)* was slightly faster than *One-Tree-kMGED(RO)* on the biggest datasets (i.e., Muta and WebPage). This confirms that the order of graphs is more important when the training database is big. Another remarkable result regarding the reordering techniques is the classification rate on Muta. The classification rate of *One-Tree-kMGED*(CAFO) on Muta is sightly higher than both *One-Tree-kMGED*(SGPCO) and *One-Tree-kMGED*(RO). To consolidate these results, the sum of the obtained distances of *One-Tree-kMGED*(CAFO) is 1.025 times more than the sum of distances of *One-Tree-kMGED*(SGPCO). This result proves that *One-Tree-kMGED*(SGPCO) obtained lower distances than *One-Tree-kMGED*(CAFO).

It is worth mentioning that a better minimization of the GED or the kMGED problems does not always lead to a higher classification rate. The GED/MGED problem is the minimization of the sum of the edit operations' costs which is different than minimizing the number of wrong classifications. However, *One-Tree-kMGED*(SGPCO) better minimized the MGED problem and provided a higher classification rate. Both criteria are not the same but are closely coupled.

Table 4: Classification results on five datasets where t refers to the average time needed by each dissimilarity computation whereas *Acc* refers to the classification accuracy. The best results are marked in bold style. Note that k was fixed to 1.

	GREC		**Protein**		**Muta**		**Fingerprint**		**WebPage**		**House-Hotel**	
	t	**Acc**	**t**	**Acc**	**t**	**Acc**	**t**	**Acc**	**t**	**Acc**	**t**	**Acc**
one-Tree-k-DF (CAFO)	136.31	**98.5**	320.77	47	70.56	**72.41**	30.25	61.68	152.48	**21**	395.66	**98,57**
one-Tree-k-DF (SGPCO)	**51.01**	**98.5**	306.48	47	**69.23**	71.05	29.06	61.68	**48.91**	**21**	370.43	**98,57**
one-Tree-k-DF (RO)	54.13	**98.5**	291.60	47	69.45	71.28	**28.76**	61.68	51.61	**21**	363.37	**98,57**
DF	491.87	**98.5**	426.90	42	487.43	70	168.45	**63.83**	470.03	**21**	485.05	**98,57**
BS-1	242.08	**98.5**	**127.35**	42.5	434.61	55.5	74.35	62.46	426.22	12.4	**108.18**	**98.57**
BS-100	293.45	58.7	475.69	31.0	486.71	55.5	211.74	10.3	499.21	4.3	478.91	**98,57**
BP	217.81	**98.5**	295.20	**52**	352.36	70	42.60	60.40	466.81	**21**	308.71	**98.57**
FBP	97.63	**98.5**	197.12	38.5	250.57	70	36.53	61.35	449.06	15	189.07	**98.57**
H	222.25	96.21	359.95	43	350.40	58.96	51.19	54.76	495.45	18.88	288.68	**98.57**

5.5.3 The impact of the training set size

To study the influence of the number of graphs in the training set on the time needed by each dissimilarity computation $DF(G_i, G_j, UB)$, we chose Muta since it is the dataset that contains the biggest number of training graphs. We tested *One-Tree-kMGED*$(SGPCO)$ and varied the number of train graphs in the dataset to see its impact on the average time needed per dissimilarity computation as depicted in Figure 10. The results shows that increasing the number of graphs decreases the average time needed for $DF(G_i, G_j, UB)$. Such results confirm that our proposed approach is more efficient when increasing the number of graphs in the training set since it helps in decreasing the value of the global UB and thus rapidly pruning the search tree.

5.5.4 The impact of the parameter k

In all the previous experiments, the value of k was fixed to 1. In this part of the experiments, we study the impact of the parameter k on the average time needed by each dissimilarity computation. As discussed in Section 4.3.1, varying k has an impact on the classification time since the obtained distance of the k^{th} nearest neighbor is used as an upper bound UB. As depicted in Figure 11, $k = 1$ represents the best case since it was the fastest in terms of the average time to classify each graph G_i. This is because the used upper bound UB is the smallest distance. Increasing the value of k increases the average time since the involved upper bound is the k^{th} one.

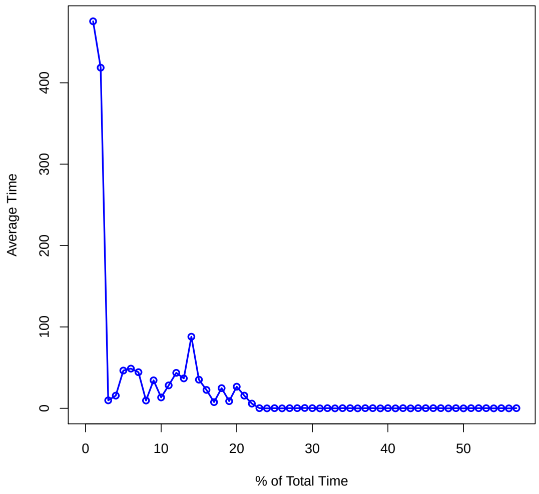

Figure 6: GREC

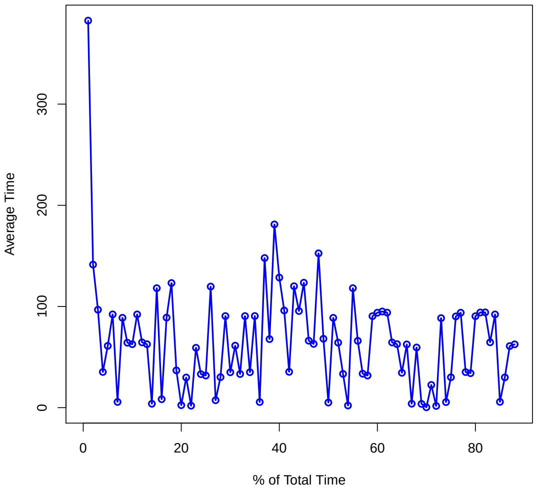

Figure 7: Muta

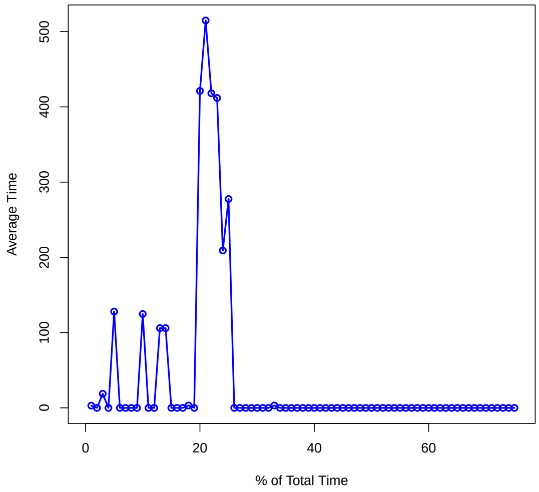

Figure 8: Fingerprint

Figure 9: *One-Tree-kMGED(RO)*: The evolution of the time needed by each dissimilarity computation while exploring the search tree of G_i.

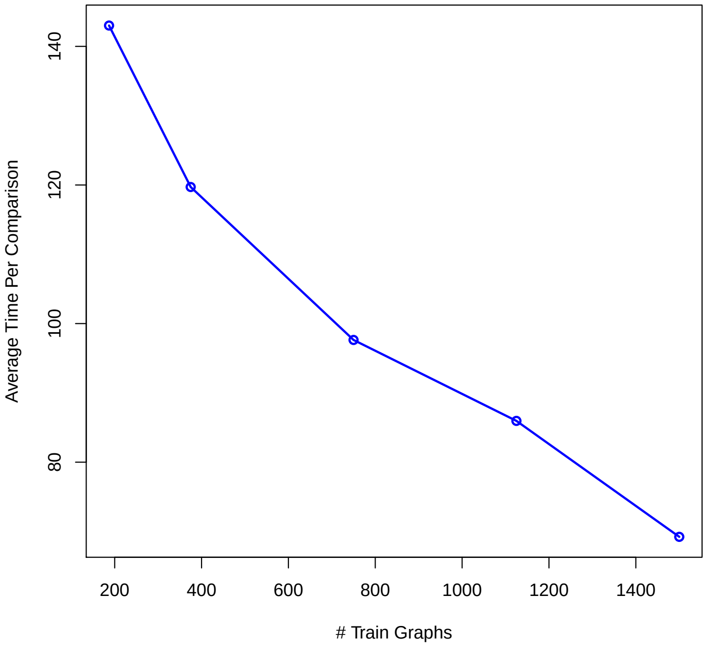

Figure 10: *One-Tree-kMGED(SGPCO)*: The impact of varying the number of training graphs on the average time needed per dissimilarity computation.

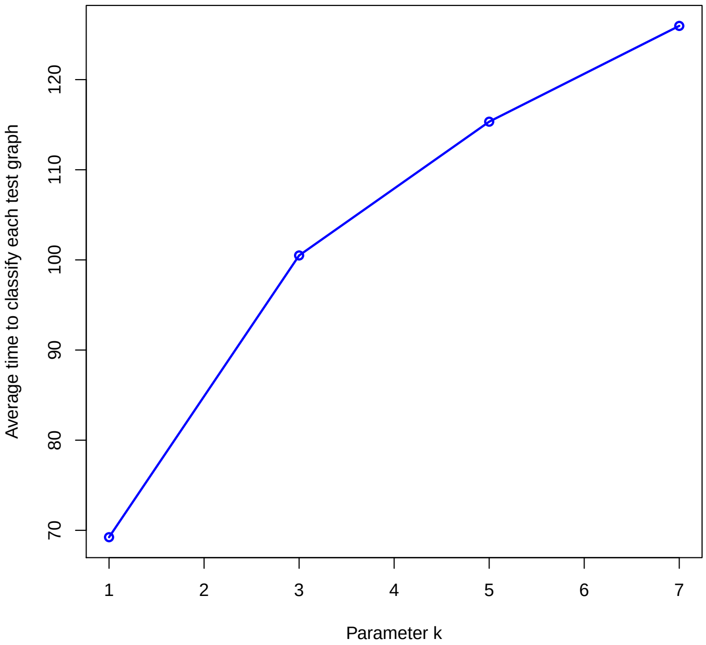

Figure 11: *One-Tree-kMGED(SGPCO) on Muta*: The impact of the parameter k on the average time needed by each dissimilarity computation.

6 Conclusions and perspectives

In this paper, we focused on the classification problem in graph space in order to keep the structural information of the graphs. The well-known k-nearest neighbors (kNN) classifier has many advantages thanks to different properties: it is non-parametric and only one parameter k is needed. However, its time consumption cannot be ignored especially when the number of graphs in the training set is big. A few research papers have proposed to shrink the number of comparisons through either clustering the training graphs or generating class prototypes. One of the main limitations of such methods is the loss of information as they imply a significant reduction of the training set (with the use of the representatives of clusters instead of all the labeled samples). Boundaries between classes can then become less precise. On the contrary, our proposed method tries to reduce time complexity by keeping all the available information. We considered the classification of a graph G_i as a single problem involving the entire training set. To do so, a new problem referred to as k-multi graph edit distance (kMGED) was defined. The problem of finding kNN falls within the kMGED problem. To propose a first algorithm for solving the new problem, a fast nearest neighbors Tree-search algorithm was put forward. This approach, referred to as *one-Tree-k-DF* takes advantage of an existing branch-and-bound based algorithm dedicated to solving the GED problem. Instead of comparing the query graph G_i with each G_j of the training set independently, *one-Tree-kMGED* groups the search trees of these comparisons inside one unique search tree dedicated to the query graph G_i. Such an approach aims at improving the upper bound as fast as possible and thus pruning the misleading parts of the global search tree. The results showed that *one-Tree-k-DF* drastically minimizes the total classification time while achieving high classification rates when compared to fast GED algorithms. The improvement of classification time was remarkably seen on Muta and Webpage, since these datasets have the largest number of training graphs. In the experiments, this fact was proved by varying the number of training graphs of Muta. The experiments were also conducted on 3 different reordering techniques of the training graphs in order to demonstrate the strong impact of the ordering of the training set. As a future work, the order of the training graphs could be learned in order to prune the search tree as fast as possible. A hierarchical representation or clustering of training graphs before the kNN stage starts could be of great interest especially when the number of training graphs is tremendous. Moreover, transforming *one-Tree-kMGED* into a parallel algorithm could reduce its computation time.

References

Abu-Aisheh, Z., Raveaux, R., Ramel, J.Y., 2016. Anytime graph matching. Pattern Recognition Letters 84, 215 – 224.

Abu-Aisheh, Z., Raveaux, R., Ramel, J.Y., 2017. Fast nearest neighbor search in graph space via a branch-and-bound strategy. GBR , 00–00.

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

Batista, G.E.A.P.A., Silva, D.F., 2009. How k-nearest neighbor parameters affect its performance, in: Argentine Symposium on Artificial Intelligence, pp. 95–106.

Bhatia, N., Vandana, 2010. Survey of nearest neighbor techniques. CoRR abs/1007.0085.

Bougleux, S., Brun, L., Carletti, V., Foggia, P., Gaüzère, B., Vento, M., 2017. Graph edit distance as a quadratic assignment problem. Pattern Recognition Letters 87, 38 – 46.

Carletti, V., Gaüzère, B., Brun, L., Vento, M., 2015. Approximate graph edit distance computation combining bipartite matching and exact neighborhood substructure distance, in: GBR, pp. 188–197.

Cover, T., Hart, P., 2006. Nearest neighbor pattern classification. IEEE Trans. Inf. Theor. 13, 21–27.

Dreiseitl, S., Ohno-Machado, L., 2002. Logistic regression and artificial neural network classification models: a methodology review. Journal of Biomedical Informatics 35, 352 – 359.

Ferrer, M., Serratosa, F., Riesen, K., 2015. A First Step Towards Exact Graph Edit Distance Using Bipartite Graph Matching. pp. 77–86.

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

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

Kooli, N., Belaïd, A., 2016. Inexact graph matching for entity recognition in ocred documents, in: 23rd International Conference on Pattern Recognition, ICPR 2016, Cancún, Mexico, December 4-8, 2016, pp. 4071–4076.

Moreno-García, C.F., Cortés, X., Serratosa, F., 2016. A graph repository for learning error-tolerant graph matching, in: SSPR, pp. 519–529.

Neuhaus, M., Riesen, K., Bunke, H., 2006a. Fast Suboptimal Algorithms for the Computation of Graph Edit Distance. pp. 163–172.

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

Raveaux, R., Adam, S., Héroux, P., Trupin, É., 2011. Learning graph prototypes for shape recognition. CVIU 115, 905–918.

Riesen, K., B.H., 2008. Iam graph database repository for graph based pattern recognition and machine learning. PRL 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., 2015a. Structural Pattern Recognition with Graph Edit Distance - Approximation Algorithms and Applications. Advances in Computer Vision and Pattern Recognition, Springer.

Riesen, K., 2015b. Structural Pattern Recognition with Graph Edit Distance - Approximation Algorithms and Applications. Advances in Computer Vision and Pattern Recognition, Springer. URL: https://doi.org/10.1007/978-3-319-27252-8, doi:10.1007/978-3-319-27252-8.

Riesen, K., Bunke, H., 2009. Graph classification based on vector space embedding. IJPRAI 23, 1053–1081.

Riesen, K., Fankhauser, S., Bunke, H., 2007. Speeding up graph edit distance computation with a bipartite heuristic., in: MLG.

Riesen, K., Fischer, A., Bunke, H., 2014. Combining Bipartite Graph Matching and Beam Search for Graph Edit Distance Approximation.

Riesen, K., et al., 2010. Graph Classification and Clustering Based on Vector Space Embedding.

Serratosa, F., 2014. Fast computation of bipartite graph matching. Pattern Recognition Letters 45, 244–250.

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

Wang, X., 2012. A fast exact k-nearest neighbors algorithm for high dimensional search using k-means clustering and triangle inequality. Proc Int Jt Conf Neural Netw 43, 2351—2358.

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.