An Exact Graph Edit Distance Algorithm for Solving Pattern Recognition Problems

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

An Exact Graph Edit Distance Algorithm for Solving Pattern Recognition Problems

Zeina Abu-Aisheh[1], Romain Raveaux[1], Jean-Yves Ramel[1] and Patrick Martineau[1]
[1]*Laboratoire d'Informatique (LI), Université François Rabelais, 37200, Tours, France*
{f_author, s_author}@univ-tours.fr

Keywords: Graph Matching, Graph Edit Distance, Pattern Recognition, Classification.

Abstract: Graph edit distance is an error tolerant matching technique 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; it represents the minimum-cost sequence of basic edit operations to transform one graph into another by means of insertion, deletion and substitution of vertices and/or edges. A widely used method for exact graph edit distance computation is based on the A* algorithm. To overcome its high memory load while traversing the search tree for storing pending solutions to be explored, we propose a depth-first graph edit distance algorithm which requires less memory and searching time. An evaluation of all possible solutions is performed without explicitly enumerating them all. Candidates are discarded using an upper and lower bounds strategy. A solid experimental study is proposed; experiments on a publicly available database empirically demonstrated that our approach is better than the A* graph edit distance computation in terms of speed, accuracy and classification rate.

1 INTRODUCTION

The comparison between two objects is a crucial operation in Pattern Recognition (PR). Representing objects by graphs turns the problem of object comparison into a graph matching one where an evaluation of structural and attributed similarity of two graphs have to be found (Vento, 2015). The similarity evaluation, called matching, is based on mapping similar vertices and similar edges of the two involved graphs.

The matching problems are all *NP-complete* except for *graph isomorphism*, for which it has not yet been demonstrated if it belongs to *NP* or not (Vento, 2015). The graph matching methods can be divided into two broad categories: *exact* graph matching and *error-tolerant* graph matching. Exact graph matching addresses the problem of detecting identical (sub)structures of two graphs g_1 and g_2 and their corresponding attributes. This category assumes the existence of only noise-free objects while in reality objects are usually affected by noise and distortion. Consequently, researchers in the PR domain often shed light on the other category, *i.e.*, inexact graph matching, for which *error-tolerance* can be easily integrated into the graph matching process.

In the context of attributed graphs, the problem of error tolerant graph matching presents a higher complexity than exact graph matching as it takes distortion and noise into account during the matching process. Indeed, the exact algorithms dedicated to solving error-tolerant graph matching are computationally complex (Vento, 2015); and (M. Neuhaus and Bunke., 2006)). Consequently, lots of works have been employed to approximately solve the error-tolerant graph matching problem. Such methods are often called heuristics or approximate methods. Approximate methods for the error-tolerant graph matching problem have been investigated based on genetic algorithm (Cross et al., 1997), probabilistic relaxation (Christmas et al., 1995), EM algorithm (Andrew D. J. Cross, 1998); (Finch et al., 1998) and neural networks (Kuner and Ueberreiter, 1988). The aforementioned techniques are expected to present a polynomial run-time. However, they cannot ensure the quality of their solutions and are likely to output suboptimal solutions.

Another error tolerant matching approach is achieved by a set of graph edit operations (*e.g.*, node insertion, node deletion, etc.). The cheapest sequence of operations needed to transform one of the two graphs into another is computed. This approach is referred to in the literature as graph edit distance. In this paper, we propose a novel graph edit distance algorithm for PR problems. Our contribution is two-fold: a depth-first graph edit distance algorithm and the choice of meaningful optimization. This proposed algorithm is supported by a solid experimental study.

The rest of the paper is organized as follows. Section 2 is devoted to the presentation of some notations and the state-of-the-art approaches for graph edit distance. Section 3 describes the phases of our approach including a new tree search exploration and a pruning strategy for exploring less number of nodes. Section 4 defines the protocol for our extensive experiments. Section 5 presents the obtained results and provides some key remarks. Finally, conclusions are drawn and future perspectives are discussed in Section 6.

2 RELATED WORKS

In this section we first define our basic notations and then introduce graph edit distance and its computation.

Definition 1. (Attributed Graph)
An attributed graph (AG) is represented by a fourtuple, AG = (V, E, μ, ζ), such that:

- V is a set of vertices.
- E is a set of edges such as $E \subseteq V \times V$.
- $\mu : V \rightarrow L_V$ is a vertex labeling function which associates label l_V to vertex v_i.
- $\zeta : E \rightarrow L_E$ is an edge labeling function which associates label l_E to edge e_i.
- L_V and L_E are vertex and edge attributes sets, respectively. These attributes can be given by a set of integers $L = \{1;2;3\}$, a vector space $L = \mathbb{R}^N$ and/or a finite set of symbolic attributes $L = \{x;y;z\}$, these sets can differ in their dimensions.

Definition 1 allows to handle arbitrarily structured graphs with unconstrained labeling functions. L_V and L_E can be represented by numeric (*e.g.*, 2D/3D points and the distances between them) and/or symbolic attributes (*e.g.*, object O_i is on top of object O_j).

2.1 Graph Edit Distance

Graph edit distance (GED) is a graph matching approach whose concept was first reported in (Sanfeliu and Fu, 1983). The basic idea of GED is to find the best set of transformations that can transform graph g_1 into graph g_2 by means of edit operations on graph g_1. The allowed operations are inserting, deleting and/or substituting vertices and their corresponding edges.

Definition 2. (Graph Edit Distance)
Let $g_1 = (V_1, E_1, \mu_1, \zeta_1)$ and $g_2 = (V_2, E_2, \mu_2, \zeta_2)$ be two graphs, the graph edit distance between g_1 and g_2 is defined as:

$$GED(g_1, g_2) = \min_{e_1, \cdots, e_k \in \gamma(g_1, g_2)} \sum_{i=1}^{k} c(e_i) \qquad (1)$$

Where c denotes the cost function measuring the strength $c(e_i)$ of an edit operation e_i and $\gamma(g_1, g_2)$ denotes the set of edit paths transforming g_1 into g_2.

A standard set of edit operations is given by insertions, deletions and substitutions of both vertices and edges. We denote the substitution of two vertices u and v by $(u \rightarrow v)$, the deletion of node u by $(u \rightarrow \varepsilon)$ and the insertion of node v by $(\varepsilon \rightarrow v)$. For edges (*e.g.*, w and z), we use the same notations as vertices. An example of an edit path between two graphs g_1 and g_2 is shown in Figure 1, the following operations have been applied in order to transform g_1 into g_2: three edge deletions, one node deletion, one node insertion, one edge insertions and three node substitutions.

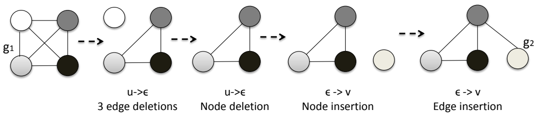

Figure 1: Transforming g_1 into g_2 by means of edit operations. Note that vertices attributes are represented in different gray scales.

Many fast heuristic methods have been proposed in the literature such as (Christmas et al., 1995); (Zeng et al., 2009); (Fankhauser et al., 2012); and (Andreas Fischer, 2013). However, these heuristic algorithms can only find unbounded suboptimal values. On the other hand, only few exact approaches have been proposed to postpone the graph size restriction (Tsai and Fu, 1979); (Justice and Hero, 2006); and (Riesen et al., 2007). Lots of exact branch and bound graph matching algorithms have been proposed in the literature. However, to the best knowledge of the authors, these works have not addressed the GED problem and cannot be easily extended to solve such a problem. For instance, a branch and bound algorithm dedicated to solving GED was proposed in (Tsai and Fu, 1979) but it was restricted to graphs that are structurally isomorphic. Afterwards, this work has been extended in (Tsai and Fu, 1983) that has taken into account insertion and deletion of nodes and edges. However, the proposed algorithm was devoted to error-correcting subgraph isomorphism.

2.2 Exact Graph Edit Distance Computation

A widely used method for exact GED computation is based on the A* algorithm (Riesen et al., 2007), this algorithm, referred to as *A*GED*, is considered as a foundation work for solving GED. *A*GED* explores the space of all possible mappings between two graphs by means of an ordered tree. Such a search tree is constructed dynamically at run time by iteratively creating successor nodes linked by edges to the currently considered node in the search tree.

Algorithm 1 Astar GED algorithm (*A*GED*)

```
Input: Non-empty attributed graphs $g_1 = (V_1,E_1,\mu_1,\nu_1)$ and $g_2 = (V_2,E_2,\mu_2,\nu_2)$ where $V_1 = \{u_1,...,u_{|v_1|}\}$ and $V_2 = \{u_2,...,u_{|v_2|}\}$
Output: A minimum cost edit path ($p_{min}$) from $g_1$ to $g_2$ e.g., $\{u_1 \to v_3,\ u_2 \to \varepsilon,\ \varepsilon \to v_2\}$
$OPEN \leftarrow \{\phi\}$, $p_{min} \leftarrow \phi$
For each node $w \in V_2$, $OPEN \leftarrow OPEN \cup \{u_1 \to w\}$
$OPEN \leftarrow OPEN \cup \{u_1 \to \varepsilon\}$
while true do
    $p_{min} \leftarrow$ argmin$\{g(p) + lb(p)\}$ s.t. $p \in OPEN$
    $OPEN \leftarrow OPEN \setminus p_{min}$
    if $p_{min}$ is a complete edit path then
        Return $p_{min}$ as a solution (i.e., the minimum cost edit distance from $g_1$ to $g_2$)
    else
        Let $p_{min} \leftarrow \{u_1 \to v_{i1},...,u_k \to v_{ik}\}$
        if $k < |V_1|$ then
            For each $w \in V_2 \setminus \{v_{i1},...,v_{ik}\}$, $OPEN \leftarrow OPEN \cup \{p_{min} \cup \{u_{k+1} \to w\}\}$
            $p_{new} \leftarrow p_{min} \cup \{u_{k+1} \to \varepsilon\}$
            $OPEN \leftarrow OPEN \cup \{p_{new}\}$
        else
            $p_{new} \leftarrow p_{min} \cup \bigcup_{w \in V2 \setminus \{v_{i1},...,v_{ik}\}} \{\varepsilon \to w\}$
            $OPEN \leftarrow OPEN \cup \{p_{new}\}$
        end if
    end if
end while
```

Algorithm 1 depicts the *A*GED* computation. In order to determine the node which is used for further expansion of the actual mapping in the next iteration, a heuristic function added to the actual partial path cost is usually used. Formally, for a node p in the search tree, $g(p)$ represents the cost of the partial edit path accumulated so far, $lb(p)$ denotes the estimated costs from p to a leaf node, $lb(p)$ must not underestimate the remaining cost in order to guarantee the optimality of the final solution. Also, it should be done in a faster way than the exact computation and return a good approximation of the true future cost. The sum $g(p) + lb(p)$ depicts the total cost assigned to an open node in the search tree. Obviously, the partial edit path p that minimizes $g(p) + lb(p)$ is chosen next for further expansion. Note that the smaller the difference between *lb* and the real future cost, the fewer the expanded nodes. The choice of *lb* is a crucial parameter and many lower bounds have been proposed in the literature. To the best of the authors' knowledge, the best lower bound has been presented in (Riesen and Bunke, 2009). In *A*GED*, *lb(p)* is computed using an assignment algorithm on unmapped vertices and edges yet to estimate the future costs. This is performed by an assignment algorithm (Riesen and Bunke, 2009) whose complexity is $O(max\{n_1, n_2\}^3)$. The estimated cost certainly constitutes a lower bound of the optimal cost as this bound represents an invalid way to edit the remaining part of g_1 into the remaining part of g_2.

2.3 Deadlocks to be Released

*A*GED* is a best-first search algorithm and so the list of candidate solutions, called *OPEN*, grows quickly. Such a fact leads to high memory consumption and thus is considered as a bottleneck of *A*GED*. In this paper, we outperform *A*GED* by getting rid of high memory consumption and the re-computation of vertices and nodes matching costs. We propose a novel algorithm that reduces the used memory space using a different exploration strategy (*i.e.,* depth-first instead of best-first). This approach also reduces the computation time as the unfruitful nodes are pruned by the lower and upper bounds strategy. A preprocessing strategy is included. First, edges and vertices costs matrices are constructed to get rid of re-computation when exploring nodes in the search tree. Second, the list V_1 is sorted to speed up the search for the best edit path to be explored.

3 OUR PROPOSAL

In order to get rid of the high memory consumption and to converge faster to the optimal solution, we propose a depth-first GED (*DF-GED*). The elements of the algorithm are described in Sections 3.1 to 3.6. Moreover, a pseudo-code is presented in Section 3.7.

3.1 Structure of Search-Tree Nodes

From now on, we will refer to the search-subtree rooted in node p as Partial Edit Path (p). Figure 2 illustrates an example of a partial edit path.

Each p is then identified by the following elements:

- *matched-vertices*(p) and *matched-edges*(p): the elements contained in these sets are vertices and edges that have been matched so far in both g_1

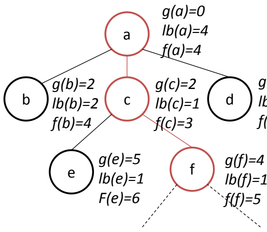

Figure 2: An example of a partial edit path p whose explored nodes so far are a, c and f. $f(*) = g(*) + lb(*)$.

and g_2. These sets can contain substitution ($u \to v$), deletion ($u \to \varepsilon$) and/or insertion ($\varepsilon \to v$) of vertices and edges, correspondingly.

- *pending-vertices*$_i(p)$ and *pending-edges*$_i(p)$: these sets represent vertices and edges of both g_1 and g_2 (*i.e.*, V_1, V_2, E_1 and E_2) that are not substituted, deleted or inserted yet where *pending-vertices*$_1(p)$ and *pending-vertices*$_2(p)$ represent pending V_1 and pending V_2, respectively whereas *pending-edges*$_1(p)$ and *pending-edges*$_2(p)$ represent pending E_1 and pending E_2, respectively.
- *parent(p)*: the parent of p.
- *children(p)*: for any node p, the exploration is achieved by choosing the next most promising vertex u_i of *pending-vertices*$_1(p)$ and matching it with all the elements of *pending-vertices*$_2(p)$ in addition to the deletion of this node (*i.e.*, $u_i \to \varepsilon$). All these matchings are referred to as *children(p)*.
- $lb(p)$: the lower bound, lb, of the estimated future cost from node p does not underestimate the complete solution. The calculation of the lower bound is described in Section 3.5.
- $g(p)$: the cost of *matched-vertices*(p) and *matched-edges*(p). Both lb and g depend on the attributes as well as the structure of the involved sub-trees. The cost functions involved with each PR dataset permit to calculate insertions, deletions and substitutions of vertices and/or edges.

3.2 Preprocessing

Preprocessing is applied before the *branch and bound* procedure starts in order to speed up the tree search exploration. First, vertices and edges cost matrices are constructed. Second, vertices-sorting is conducted.

3.2.1 Cost Matrices

The vertices and edges cost matrices (C_v and C_e) are constructed, respectively. This step aims at speeding up branch and bound by getting rid of re-calculating the assigned costs 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 $V2 = (v_1, ..., v_m)$. A vertices cost matrix C_v, whose dimension is $(n+2)$ X $(m+2)$, is constructed as follows:

$$C_v = \left|\begin{array}{cccc||cc}
c_{1,1} & \cdots & \cdots & c_{1,m} & c_{1\leftarrow\varepsilon} & c_{1\rightarrow\varepsilon} \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
c_{n,1} & \cdots & \cdots & c_{n,m} & c_{n\leftarrow\varepsilon} & c_{n\rightarrow\varepsilon} \\
\hline
c_{\varepsilon\rightarrow 1} & \cdots & \cdots & c_{\varepsilon\rightarrow m} & \infty & \infty \\
c_{\varepsilon\leftarrow 1} & \cdots & \cdots & c_{\varepsilon\leftarrow m} & \infty & \infty
\end{array}\right|$$

where n is the number of vertices of g_1 and m 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 $\varepsilon - \varepsilon$.

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.

3.2.2 Vertices-Sorting Strategy

As *GED* aims at transforming g_1 into g_2 so it is important to sort V_1 in order to start with the most promising vertices that will speed up the exploration of the search tree while searching for the optimal solution. The aforementioned C_v is used as an input of the vertices-sorting phase. To sort V_1, Munkres' algorithm is applied. From now on, we will refer to the set of sorted vertices as *sorted*-V_1.

3.3 Branching Strategy

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 node 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.

The root r of the search-tree is the node with *matched-vertices*(r) = $\{\phi\}$, *matched-edges*(r) = $\{\phi\}$, *pending-vertices*$_{1,2}(r) = V_1 \cup V_2$, *pending-edges*$_{1,2}(r) = E_1 \cup E_2$, $g(r) = \infty$ and $lb(r) = \infty$. Initially r is the only node in the 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 root's children *children(r).* Then, *children(r)* is added to *OPEN*. Consequently, a minimum edit path (p_{min}) is chosen to be explored by selecting the minimum cost node (*i.e.*, $min(g(p)+lb(p))$) among *children(r)* and so on. We backtrack to continue the search for a good edit path by revoking p_{min} (if p_{min} equals ϕ) and trying out the next child in the set of *children(r)* and so on.

3.4 Reduction Strategy

As in *A*GED*, pruning, or bounding, is achieved thanks to $lb(p)$, $g(p)$ and a global upper bound UB obtained at node leaves. Formally, for a node p in the search tree, the sum $g(p)+lb(p)$ is taken into account and compared with UB. That is, if $g(p)+lb(p)$ 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 *DF-GED*. This algorithms differs from *A*GED* as at any time t, in the worst case, *OPEN* contains approximately $|V_1|.|V_2|$ elements and hence the memory consumption is not exhausted.

3.5 Lower Bound

The lower bound $lb(p)$, adapted to *DF-GED*, is the one used in section *A*GED*, see Section 2.2.

3.6 Upper Bound

The very first upper bound (UB) is computed by Munkres' algorithm as it provides reasonable results, see (Riesen and Bunke, 2009) for more details. Afterwards and while traversing the search tree, UB is replaced by the best UB found so far (*i.e.*, a complete path whose cost is less than the current UB). After finishing the traversal of the search tree (*i.e.*, when p_{min} equals ϕ and its parent is r), the best UB is outputted as an optimal solution of *DF-GED*. Encountering upper bounds when performing a depth-first traversal efficiently prunes the search space and thus helps at finding the optimal solution faster than *A*GED*.

3.7 Pseudo Code

As depicted in Algorithm 2, *DF-GED* starts by a preprocessing step (line 2), then an upper bound UB is calculated by Munkres' algorithm (line 3). The traversal of the search tree starts by selecting a first vertex $u_1 \in$ *sorted*-V_1 where u_1 substituted with all vertex w in graph g_2 as well as the deletion case ($u_1 \rightarrow \varepsilon$) are inserted into *OPEN* (lines 4 and 5). A branching step is performed in line 8 where the best child p_{min} is selected, the backtracking is done when there is no more children to explore in the selected branch and the parent node ($parent_{tmp}$) is not the root (lines 9 to 12). p_{min} is explored by substituting the next promising node u_{k+1} with *pending-vertices*$_2(p_{min})$ and also deleting u_{k+1}, respectively (lines 18 to 24). Similar to *A*GED*, if *pending-vertices*$_1(p_{min})$ equals ϕ, *pending-vertices*$_2(p_{min})$ will be inserted (line 26). UB and *Best-Edit-Path* are updated whenever a better UB is encountered (line 28). Each time the next parent to be explored will be replaced by p_{min} (line32). This algorithm guarantees to find the optimal solution of $GED(g_1,g_2)$. Note that edges operations are taken into account in the matching process when substituting, deleting or inserting their corresponding vertices.

4 EXPERIMENTS

4.1 Environment

Evaluations are conducted on a 4-core Intel i7 processor 3.07GHz and 8 GB of memory.

4.2 Compared Methods

We compare *A*GED* with *DF-GED*. Both methods return a node to node matching.

4.3 Database in Use

To the best of our knowledge, few publicly available graphs databases are dedicated to 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 and Bunke, 2008). Moreover, it contains graphs of both symbolic and numeric attributes which is not often the case of other datasets.

The GREC database used in our experiments is a dataset in IAM which consists of a subset of the symbol database underlying the GREC2005 competition (Dosch and Valveny, 2006). The images of GREC represent symbols from architecture, electronics, and

Algorithm 2 Depth-first GED algorithm (*DF-GED*)

```
Input: Non-empty attributed graphs $g_1 = (V_1,E_1,\mu_1,\nu_1)$ and $g_2 = (V_2,E_2,\mu_2,\nu_2)$ where $V_1 = \{u_1,...,u_{|v_1|}\}$ and $V_2 = \{u_2,...,u_{|v_2|}\}$
Output: A distance $UB$ and a minimum cost edit path (Best-Edit-Path) from $g_1$ to $g_2$ e.g., $\{u_1 \rightarrow v_3, u_2 \rightarrow \varepsilon, \varepsilon \rightarrow v_2\}$
$OPEN \leftarrow \{\phi\}$, $Best\text{-}Edit\text{-}Path \leftarrow \phi$
Generate $C_v$, $C_e$ and $sorted\text{-}V_1$
$UB \leftarrow$Munkres($g_1$,$g_2$)
For each node $w \in V_2$, $OPEN \leftarrow OPEN \cup \{u_1 \rightarrow w\}$ s.t. $u_1 \in sorted\text{-}V_1$
$OPEN \leftarrow OPEN \cup \{u_1 \rightarrow \varepsilon\}$
$p_{min} \leftarrow \phi$, $r \leftarrow$ parent($u_1$), $parent_{tmp} \leftarrow r$
while true do
    $p_{min} \leftarrow$ bestChild($parent_{tmp}$)
    while $p_{min} == \phi$ and $parent_{tmp} != r$ do
        $parent_{tmp} \leftarrow$ backtrack($parent_{tmp}$)
        $p_{min} \leftarrow$ bestChild($parent_{tmp}$)
    end while
    if $p_{min} == \phi$ and $parent_{tmp} == r$ then
        Return $UB$, $Best\text{-}Edit\text{-}Path$
    end if
    $OPEN \leftarrow OPEN \setminus p_{min}$
    if $g(p_{min}) + lb(p_{min}) < UB$ then
        if $pending\text{-}vertices_1(p_{min})$!= $\{\phi\}$ then
            for $w \in pending\text{-}vertices_2(p_{min})$ do
                $p \leftarrow p_{min} \cup \{u_{k+1} \rightarrow w\}$
                If $g(p) + lb(p) < UB$ then $OPEN \leftarrow OPEN \cup \{p\}$
            end for
            $p \leftarrow p_{min} \cup \{u_{k+1} \rightarrow \varepsilon\}$
            If $g(p) + lb(p) < UB$ then $OPEN \leftarrow OPEN \cup \{p\}$
        else
            Generate a complete solution $p \leftarrow p_{min} \cup \bigcup_{w \in pending-vertices_2(p)} \{\varepsilon \rightarrow w\}$
            if $g(p) + lb(p) < UB$ then
                $UB \leftarrow g(p)$, $Best\text{-}Edit\text{-}Path \leftarrow p$
            end if
        end if
    end if
    $parent_{tmp} \leftarrow p_{min}$
end while
```

other technical fields. Distortion operators are applied to the original images in order to simulate handwritten symbols. To this end, the primitive lines of the symbols are divided into subparts. The ending points of these subparts are then randomly shifted within a certain distance, maintaining connectivity. Each node represents a subpart of a line and is attributed with its relative length (ratio of the length of the actual line to the length of the longest line in the symbol). Connection points of lines are represented by edges attributed with the angle between the corresponding lines. The dataset consists of 6 classes and 30 instances per class. This dataset is useful as both vertices and edges are labeled with numeric attributes. In addition, it holds graphs whose sizes vary from 5 to 25 vertices.

Cost Function: The vertices of graphs from GREC dataset are labeled with (x, y) coordinates and a type (ending point, corner, intersection or circle). The same accounts for the edges where two types (line, arc) are employed. The Euclidean cost model is adopted accordingly. That is, for node substitutions the type of the involved vertices is compared first. For identically typed vertices, the Euclidean distance is used as node substitution cost. In case of non-identical types on the vertices, the substitution cost is set to $2\tau_{node}$, which reflects the intuition that vertices with different type label cannot be substituted but have to be deleted and inserted, respectively. For edge substitutions, the dissimilarity of two types is measured with a Dirac function returning 0 if the two types are equal, and $2\tau_{edge}$ otherwise. Both τ_{node} and τ_{edge} are non-negative parameters. To control whether the edit operation cost on the vertices or on the edges is more important another parameter α is integrated. In our experiments we have set τ_{node}, τ_{edge} and α to 90, 15 and 0.5, respectively. These meta parameters' values are taken from (Riesen and Bunke, 2010). This dataset consists of 1,100 graphs where graphs are uniformly distributed between 22 symbols. The resulting dataset is split into a training and a validation set of size 286 each, and a test set of size 528.

GREC Decomposition: We decompose the database into subsets, each of which contains graphs that have the same size. We focus on the following subsets: (GREC5, GREC10, GREC15 and GREC20) aiming at evaluating the two methods when increasing the size of the involved graphs (*i.e.*, the number of vertices). Due to the large number of matchings considered and the exponential complexity of the tested algorithms, we select 10 graphs of the train set of each subset of GREC. The train set is representative of all graph distortions and the selection of only 10 graphs per subset ends up having 100 pairwise comparisons which is significant for such a kind of experiments. Furthermore, we add another subset GREC-mix that contains different number of vertices (*i.e.*, 10 graphs with different number of vertices).

4.4 Protocol and Quality Measures

Our Protocol is two-fold. First, calculating the *distance matrix* under small and big time constraints. Second, Classification test under a reasonable time constraint.

Let $\mathcal{S}$ be a graph dataset consisting of m graphs, $\mathcal{S} = \{g_1, g_2, ..., g_m\}$. Let $\mathcal{P} = \{\text{A*GED, DF-GED}\}$ be the set the compared methods. Given a method $p \in \mathcal{P}$, we computed the square distance matrix $M^p \in \mathcal{M}^m(\mathbb{R}^+)$, that holds every pairwise comparison $M^p_{i,j} = d_p(g_i, g_j)$, where the distance $d_p(g_i, g_j)$ is the value returned by the method p on the graph pair (g_i, g_j) within a certain time and memory limits. Hence, $M^{\text{A*GED}}$ and $M^{\text{DF-GED}}$ denote distance matrices of *A*GED* and *DF-GED* methods, respectively.

Let $GT \in \mathcal{M}^m(\mathbb{R}^+)$ be the reference matrix that holds the best found distance for each pair of graphs. We aim at comparing the errors committed by the two methods as well as their running time under a time constraint C_T and a memory constraint C_M when graphs' sizes increase (*i.e.*, on GRECk) and also on *GREC-mix*. To this objective, we test the accuracy of $\mathcal{P}$ when C_T is small, *e.g.,* C_T = 350 milliseconds (ms), and when C_T is big (*e.g.,* C_T = 5 minutes). C_M is set to 1GB during all the experiments. We expect *A*GED* to violate C_M specially when graphs get larger.

In the following, we define the measurements used for evaluating our protocol:

Deviation: We evaluate the error committed by a method p over the reference distances. To this end, we measure an indicator called deviation and defined by the following equation:

$$deviation(i,j)^p = \frac{|M^p_{i,j} - GT_{i,j}|}{GT_{i,j}}, \ \forall (i,j) \in [\![1,m]\!]^2 \tag{2}$$

Where $GT_{i,j}$ is the smallest distance among all distances generated by p when matching g_i and g_j.

Running Time: We measure the running time in milliseconds for each comparison $d(g_i; g_j)$. This value reflects the overall time for GED computation including all the inherits costs computations (*i.e.*, $g(p)$; $lb(p)$; and UB).

For the classification test, we are interested in the average computation time (t) which corresponds to the average time elapsed when classifying all the test graphs of GREC and the classification accuracy (AC) which defines the error made when classifying the test graphs of GREC. Both measurements are achieved when C_T = 500ms and C_M = 1GB. The classification stage is performed by a *1*-NN classifier. Each test graph g_t is compared to the entire training set. The nearest neighbor's label is assigned to g_t.

5 RESULTS AND DISCUSSION

Figure 3 shows the deviation results under 350 ms and 5 minutes respectively. We observe that *DF-GED* always outperforms *A*GED* under the same time constraint. For all GED computations, *DF-GED* gives the best distance (*i.e.*, $GT_{i,j}$) and so its deviation is always 0%. In contrast with *DF-GED*, the deviation of *A*GED* decreases when C_T = 5 minutes. However, when the size of graphs increase (*e.g.*, GREC15 and GREC20), the deviation starts to converge due to memory saturation where the best recently known solution is outputted before halting.

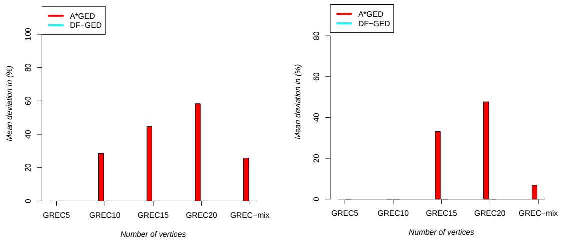

Figure 3: Deviation: Left (350 ms), Right (5 minutes).

Figure 4 demonstrates the running time of both *A*GED* and *DF-GED*. When C_T = 350 ms both running times are relatively equal on GREC15 and GREC20 and that is because non of them is able to find an optimal solution before exceeding C_T. When C_T increases, *DF-GED* becomes faster as it explores the search tree in a depth-first way (*i.e.*, not stopped by C_M) while pruning the search tree thanks to its upper and lower bounds as well as the preprocessing step, see Section 3.2. On the other hand, *A*GED* does not continue for further exploration on GREC15 and GREC20 because of the size of the involved graphs where available memory is exhausted and so the best recently known solution is given before halting.

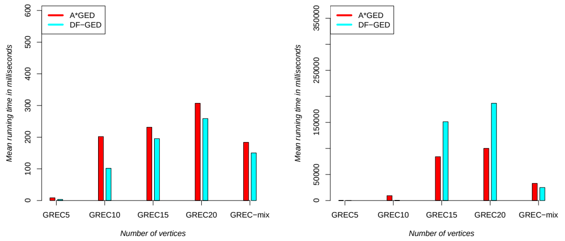

Figure 4: Running Time: Left (350 ms), Right (5 minutes).

For the classification experiment, 151008 comparisons are performed on 286 training graphs and 528 test graphs of the GREC dataset. As depicted in Table 1, results show that the classification accuracy AC of *DF-GED* is 2.3 times higher under the same C_T where C_T=500 ms. Moreover, *DF-GED* is 1.9 times faster as the average time t is smaller.

Algorithms	t	AC
*A*GED*	119491.5 ms	42.23%
DF-GED	60468.7 ms	98.48 %

Table 1: Classifying graphs of GREC (C_T = 500 ms).

6 CONCLUSION AND PERSPECTIVES

In the present paper, we have considered the problem of GED computation for PR. Graph edit distance is a powerful and flexible paradigm that has been used in different applications in PR. The exact algorithm, *A*GED*, presented in the literature suffers from high memory consumption and thus is too costly to match large graphs. In this paper, we propose another exact GED algorithm, *DF-GED*, which is based on depth-first search. This algorithm speeds up the computations of graph edit distance 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 in the set *OPEN* is relatively small thanks to the depth-first search where the number of pending nodes is $|V1|.|V2|$ in the worst case.

In the experimental section, we have proposed to evaluate sub-optimally both exact methods: *A*GED* and *DF-GED* under some memory and time constraints. Experiments on the GREC database empirically demonstrated that *DF-GED* outperforms *A*GED* in terms of precision, speed and classification rate. In future work, we aim at proposing a benchmark to measure the quality of the solutions found by approximate methods. Both exact and approximate graph edit distance computations will be evaluated on different PR databases.

REFERENCES

Andreas Fischer, Ching Y. Suen, V. F. K. R. H. B. (2013). A fast matching algorithm for graph-based handwriting recognition. *GbRPR 2013*, 7877:194–203.

Andrew D. J. Cross, E. R. H. (1998). Graph matching with a dual-step em algorithm. *IEEE Trans. Pattern Anal. Mach. Intell.*, 20(11):1236–1253.

Christmas, W., Kittler, J., and Petrou, M. (1995). Structural matching in computer vision using probabilistic relaxation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 17(8):749–764.

Cross, A. D., Wilson, R. C., and Hancock, E. R. (1997). Inexact graph matching using genetic search. *Pattern Recognition*, 30(6):953–970.

Dosch, P. and Valveny, E. (2006). Report on the second symbol recognition contest. 3926:381–397.

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

Finch, A., Wilson, R., and Hancock, E. (1998). An energy function and continuous edit process for graph matching. *Neural Computation*, 10(7):1873–1894.

Justice, D. and Hero, A. (2006). A binary linear programming formulation of the graph edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(8):1200–1214.

Kuner, P. and Ueberreiter, B. (1988). Pattern recognition by graph matching: Combinatorial versus continuous optimization. *International Journal in Pattern Recognition and Artificial Intelligence*, 2(3):527–542.

M. Neuhaus, K. R. and 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.

Riesen, K. and Bunke, H. (2008). Iam graph database repository for graph based pattern recognition and machine learning. pages 287–297.

Riesen, K. and Bunke, H. (2009). Approximate graph edit distance computation by means of bipartite graph matching. *Image and Vision Computing*, 27(7):950–959.

Riesen, K. and Bunke, H. (2010). *Graph Classification and Clustering Based on Vector Space Embedding*.

Riesen, K., Fankhauser, S., and Bunke, H. (2007). Speeding up graph edit distance computation with a bipartite heuristic. In *Mining and Learning with Graphs, MLG 2007, Proceedings*.

Sanfeliu, A. and Fu, K. (1983). A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, 13(3):353–362.

Tsai, W.-H. and Fu, K.-S. (1979). Error-correcting isomorphisms of attributed relational graphs for pattern analysis. *Systems, Man and Cybernetics, IEEE Transactions on*, 9(12):757–768.

Tsai, W.-H. and Fu, K.-S. (1983). Subgraph error-correcting isomorphisms for syntactic pattern recognition. *Systems, Man and Cybernetics, IEEE Transactions on*, SMC-13(1):48–62.

Vento, M. (2015). A long trip in the charming world of graphs for pattern recognition. *Pattern Recognition*, 48(2):291–301.

Zeng, Z., Tung, A. K. H., Wang, J., Feng, J., and Zhou, L. (2009). Comparing stars: On approximating graph edit distance. *Proc. VLDB Endow.*, 2(1):25–36.