Distributed Graph Matching and Graph Indexing Approaches
Applications to Pattern Recognition

Zeina Abu-Aisheh, Romain Raveaux and Jean-Yves Ramel
Laboratoire d'Informatique (LI), Université François Rabelais, 37200, Tours, France

Keywords: Graph Matching, Graph Edit Distance, Pattern Recognition, Classification, Distributed Systems, Scalability.

Abstract: Attributed graphs are powerful data structures for the representation of complex objects. In a graph-based representation, vertices and their attributes describe objects (or part of objects) while edges represent interrelationships between the objects. Due to the inherent genericity of graph-based representations, and thanks to the improvement of computer capacities, structural representations have become more and more popular in the field of Pattern Recognition (PR). In this thesis, we tackle two important graph-based problems for PR: Graph Matching and Graph Indexing. The comparison between two objects is a crucial operation in PR. Representing objects by graphs turns the problem of object comparison into graph matching where correspondences between nodes and edges of two graphs have to be found. Moreover, graph-based indices are important so that a graph query can be retrieved from a large database via such indices, such a problem is referred to as *graph indexing.* The complexity of both graph matching and graph indexing is generally stated to be NP-COMPLETE or NP-hard. Coming up with a graph matching algorithm that can scale up to match graphs involved in PR tasks is a great challenge. Among the graph matching methods dedicated to PR problems, the Graph Edit Distance (GED) is of great interest. Over the last decade, GED has been applied to a wide range of specific applications from molecule recognition to image classification. In this report, we present the first part of the thesis. We tackle GED, shed light on the importance of having exact solutions rather than approximate ones and come up with a distributed GED where the search tree is decomposed into smaller trees which are solved independently and in a complete distributed manner. In the second part of the thesis, we aim at proposing new distributed graph-indexing approaches that aim at retrieving a graph from a large graph-based index as fast as possible. *Graph indexing* will be reported as a perspective of this work.

1 INTRODUCTION

Graphs are an efficient data structure for structural objects representation. Over the last decades, much attention has been shed on using graphs as a structural representation of objects in Pattern Recognition (PR).

In PR, the attributes of both nodes and edges play an important role for representing graphs. Combining both symbolic and numeric attributes on nodes and edges makes the extracted nodes and edges more meaningful and highly representative. Unlike the other graphs used in other fields (*e.g.,* shortest path) where the combination of symbolic and numeric attributes is not necessarily needed. Representing objects by graphs easily transforms the problem of object detection or objects comparison into a graph matching one.

Graph matching is the process of finding a correspondence between the nodes and the edges of two graphs g_1 and g_2 that satisfies some (more or less stringent) constraints ensuring that similar substructures of the source graph are mapped to similar substructures of the target graph. Recently, graph matching has been considered as a fundamental problem in PR. Such a problem is known to be NP-complete, except for graph isomorphism, for which it has not yet been demonstrated whether it belongs to NP or not (Conte et al., 2004).

Exact graph matching addresses the problem of detecting identical (sub)structures of two graphs g_1 and g_2 and their corresponding labels. This category assumes the existence of only noise-free objects while in reality objects are usually affected by noise and distortion. Consequently, researchers often shed light on the other category, *i.e.*, inexact or so-called error-tolerant graph matching, where some degree of 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

Abu-Aisheh Z., Raveaux R. and Ramel J..
Distributed Graph Matching and Graph Indexing Approaches - Applications to Pattern Recognition.
Copyright © 2015 SCITEPRESS (Science and Technology Publications, Lda.)

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, 2014); 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 algrithm (A.D.J. Cross and Hancock, 1997), probabilistic relaxation (W. Christmas and Petrou., 1995), EM algorithm (Andrew D. J. Cross, 1998); (Finch, 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.

Graph Edit Distance, referred to as GED, is an error-tolerant technique that has been widely studied and largely applied to PR (Tsai and Fu, 1979). Its flexibility comes from its generality as it is applicable on unconstrained attributed graphs. GED is a generalization of the graph isomorphism problem where the goal is to minimize the cost of graph transformation. In GED, graph g_1 is transformed into graph g_2 by means of series of transformations. The allowed edit operations are: deletion, insertion and substitution of nodes and their corresponding edges. GED is computationally complex or expensive, it is said to be an NP-COMPLETE problem where the complexity is exponential in the number of nodes of the involved graphs. Such a fact limits GED algorithms to work on relatively small graphs. To overcome this problem three main directions have been adopted in the literature. First, optimal methods based on admissible heuristics to prune the search space (*e.g.*, (Riesen et al., 2007)). Second, sub-optimal methods simplifying the problem (*e.g.,* (M. Neuhaus and Bunke., 2006)). Third, sub-optimal methods by means of approximate optimization algorithm (*e.g.*, (Riesen, 2009); and (Andreas Fischer, 2013)). However, sub-optimal methods does not guarantee to find the best matching and the error rate gets higher as the involved graphs get larger. Accordingly, in this thesis a focus is given to optimal methods. To prevent the combinatorial explosion, many works have been focused on efficiently pruning the search space. The computation of admissible lower bounds have been deeply studied to reduce memory and CPU complexity (Bunke, 1983); and (Zeng et al., 2009).

Most of the current techniques are optimized for centralized graph processing. A distributed approach providing horizontal scalability is required in order to handle the analysis workload. Thus, besides providing lower and upper bounds for the problem, we have adopted the idea of decomposing GED into smaller problems, or sub-problems, via a divide and conquer strategy. The sub-problems are then solved in a distributed manner.

The rest of this report is organized as follows. In Section 2, a focus on the related works is given. In Section 3 the notations and the definitions used in the paper are presented and our approach is positioned up in the literature. Section 4 reports our proposed sequential algorithm used to solve GED. In Section 5, the chosen parallel computing model and the proposed distributed GED are presented, respectively. In Section 6 the databases and the experimental protocol used to point out the performance of the proposed approaches are represented. Section 7 demonstrates the results achieved so far. Finally, conclusions are drawn and future perspectives are discussed in Section 8.

2 RELATED WORKS

The distributed and parallel graph matching methods, presented in the literature, can be divided into two categories: *Data-Parallelism and Search-Parallelism.* In Data-Parallelism, the graphs g_1 and g_2 can be partitioned into sub-graphs. These small sub-graphs can be matched independently in a sequential or in a parallel manner. The results of all the sub-problems are reassembled producing a global answer of the main graph matching problem ((Qiu and Hancock, 2006); (Patwary et al., 2010); and (Kollias, 2012)). In Search-Parallelism, matching g_1 and g_2 is considered as a single problem. However, the search space of g_1 and g_2 is partitioned and then processed in a completely parallel manner ((Allen and Yasuda, 1997); (Wan, 1998); and (Plantenga, 2013)).

We focus on two distributed works, belonging to the Search-Parallelism category and are dedicated to solving graph matching problems:

2.1 Maspar-SIMD

A parallel inexact graph matching algorithm is proposed in [22]. This algorithm, referred to as MasPar-SIMD, is depth-first branch-and-bound for determining a minimum-distance correspondence between two unlabeled graphs. The heuristic, called *forward checking*, used to prune the search space, examines the possible sources of edges mismatches and thus forward checking keeps track of constraints (edges) that are not satisfied. The degree of mismatch of a

SCIENCE AND TECHNOLOGY PUBLICATIONS

permutation π is defined of as the number of edges in g_1 that are not mapped by π to edges in g_1 plus the number of edges in g_2 not mapped to edges in g_1. At each iteration, the edge errors are accumulated. Results show that MasPar-SIMD consistently outperforms the sequential version when the involved graphs have more than 16 vertices. This algorithm is inexact for two reasons. First, the mismatch of edges are taken into account whereas mismatch of nodes are ignored. Second, the heuristic is not a lower bound and cannot ensure the optimal solution to be found. Also, the graphs involved in the algorithm are unlabeled and thus numeric and attributed graphs are not included. As for the exploration of the search space, the best permutation and the best degree of match are only updated at the end of each iteration, such a fact does not allow pruning the search space as fast as possible.

2.2 SGIA-MR

Recently, a *Hadoop MapReduce* implementation for subgraph type isomorphism, called *SGIA-MR*, has been implemented in (Plantenga, 2013). This algorithm makes use of an implicit multi-partite graph which finds sub-graphs in a single large graph (*i.e.*, given a pattern G_p, find all the walks in the graph G that correspond to the pattern G_p).

In (Plantenga, 2013), the type-isomorphism match is inexact in the sense that matched subgraphs do not necessarily have isomorphic structures. *SGIA-MR* only considers symbolic or unlabeled graphs and thus it cannot be easily extended to work on numeric attributed graphs. Moreover, this algorithm is a noise-free one as neither nodes nor edges distortion is included in the matching process.

2.3 Conclusion on Parallel and Distributed Graph Matching Methods

To our best of knowledge, none of the parallel and distributed graph matching methods has included numeric attributed graphs, besides that all of them fall within the suboptimal graph matching category and so they cannot ensure the optimal solution to be found. Moreover, neither bounds on the optimal solution nor quality measure or confidence on the solution are provided. In the literature, there is also no effort devoted to solving GED problems in a distributed manner. Indeed, MasPar-SIMD and SGIA-MR cannot be easily extended to solve GED. We believe that there is a need for a distributed and optimal GED method that can be used to match symbolic and numeric attributed graphs in a way that outperforms the sequential GED algorithms.

3 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 1. 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:

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

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 used for vertices.

A complete edit path (EP) refers to an edit path that fully transforms g_1 into g_2 (*i.e.,* complete solution). Mathematically, $EP = \{e_i\}_{i=1}^{k}$. 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.

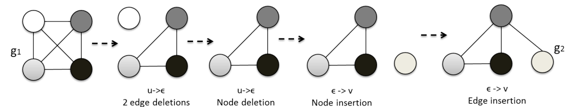

Figure 1: Transforming g_1 into g_2 by means of edit operations. Note that vertices attributes are represented in different gray scales.

Many fast heuristic methods have been proposed in the literature such as (W. Christmas and Petrou., 1995); (Zeng et al., 2009); (Fankhauser et al., 2012); (Combier et al., 2013); 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.

3.1 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. In the worst case, the space complexity can be expressed as $O(|\sigma|)$ (Cormen et al., 2009) where $|\sigma|$ is the cardinality of the set of all possible edit paths. Since $|\sigma|$ is exponentional in the number of vertices involved in the graphs, the memory usage is still an issue.

Algorithm 1 depicts the *A*GED* computation. In order to determine the node which will be used for further expansion of the actual mapping in the next iteration, a heuristic function added to the actual edit path cost is usually used. Formally, for a node p in the search tree, $g(p)$ represents the cost of the edit path accumulated so far, $h(p)$ denotes the estimated costs from p to a leaf node, $h(p)$ 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 $g(p)+h(p)$, referred to as $lb(p)$, depicts the total cost assigned to an open node in the search tree. Obviously, the edit path p that minimizes $g(p)+h(p)$ 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, 2009). In *A*GED*, *h(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, 2009) whose complexity is $O(max\{n_1,n_2\}^3)$. The unprocessed edges of both graphs are handled analogously. Obviously, this procedure allows multiple substitutions involving the same vertex or edge and, therefore, it possibly represents an invalid way to edit the remaining part of g_1 into the remaining part of g_2. However, the estimated cost certainly constitutes a lower bound of the optimal cost.

Algorithm 1 : Astar Graph Edit Distance algorithm (*A*GED*).

Input: Non-empty attributed graphs $g_1 = (V_1,E_1,\mu_1,v_1)$ and $g_2 = (V_2,E_2,\mu_2,v_2)$ where $V_1 = \{u_1,...,u_{|v_1|}\}$ and $V_2 = \{u_2,...,u_{|v_2|}\}$
Output: A minimum cost edit path from g_1 to g_2 *e.g.*, p_{min}=$\{u_1 \to v_3,\ u_2 \to \varepsilon\ ,\ \varepsilon \to v_2\}$

1: Initialize *OPEN* to the empty set
2: For each node $w \in V_2$, insert the substitution insert the substitution $\{u_1 \to w\}$ into *OPEN*
3: Insert the deletion $\{u_1 \to \varepsilon\}$ into *OPEN*
4: **while** true **do**
5: Find p_{min} = argmin$\{g(p)+h(p)\}$ s.t. $p \in$ *OPEN*
6: Remove p_{min} from *OPEN*
7: **if** p_{min} is a complete edit path **then**
8: Return p_{min} as the solution (*i.e.*, a minimum cost edit distance from g_1 to g_2)
9: **else**
10: Let $p_{min} = \{u_1 \to v_{i1},...,u_k \to v_{ik}\}$
11: **if** $k < |V_1|$ **then**
12: For each $w \in V_2 \setminus \{v_{i1},...,v_{ik}\}$, insert $p_{min} \cup \{u_{k+1} \to w\}$ into *OPEN*
13: $p_{new} = \{p_{min}, u_{k+1} \to \varepsilon\}$
14: Insert p_{new} into *OPEN*
15: **else**
16: Insert $p_{min} \cup \bigcup_{w \in V2 \setminus \{v_{i1},...,v_{ik}\}} \{\varepsilon \to w\}$ into *OPEN*
17: **end if**
18: **end if**
19: **end while**

3.2 Conclusion on *A*-GED*

*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 new al-

gorithm 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.

4 OUR SEQUENTIAL PROPOSAL

In this section, we introduce our proposal and we mention its advantages over *A*GED*.

4.1 Depth-first Graph Edit Distance

As mentioned before, $A*$, with all its different lower bounds, suffers from a high memory consumption while searching for the best edit path. $A*$, which is a best-first search algorithm, constructs the search tree dynamically at run time. The construction of the search tree is achieved iteratively by creating successor nodes linked by edges to the currently considered node in the search tree.

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*). This strategy has O($|V_1.V_2|$) space complexity in the worst case for storing the pending edit paths in the set *OPEN* (Cormen et al., 2009). That is, at any time t the number of pending nodes is relatively small and thus there is no high memory consumption as in $A*$.

The elements of the algorithm are described in Sections 4.2 to 4.7. Moreover, a pseudo-code for the method is presented in Section 4.8.

4.2 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:

- $matched\text{-}vertices(p)$ and $matched\text{-}edges(p)$: the elements contained in these sets are vertices and edges that have been matched so far in both g_1 and g_2. These sets can contain substitution ($u \rightarrow v$), deletion ($u \rightarrow \varepsilon$) and/or insertion ($\varepsilon \rightarrow v$) of vertices and edges, correspondingly.

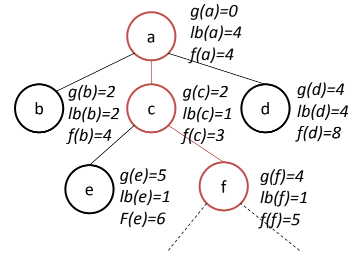

Figure 2: An example of a partial edit path p whose explored nodes so far are a, c and f. $f(*) = g(*) + h(*)$.

- $pending\text{-}vertices_i(p)$ and $pending\text{-}edges_i(p)$: 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 $pending\text{-}vertices_1(p)$ and $pending\text{-}vertices_2(p)$ represent pending V_1 and pending V_2, respectively whereas $pending\text{-}edges_1(p)$ and $pending\text{-}edges_2(p)$ represent pending E_1 and pending E_2, respectively.
- *parent(p)*: the parent of p.
- *siblings(p)*: for any node p, the exploration is achieved by choosing the next most promising vertex u_i of $pending\text{-}vertices_1(p)$ and matching it with all the elements of $pending\text{-}vertices_2(p)$ in addition to the deletion of this node (*i.e.,* $u_i \rightarrow \varepsilon$). All these matchings are referred to as *siblings(p)*.
- $h(p)$: is the estimated future cost from node p. $h(p)$ does not underestimate the complete solution. The calculation of $h(p)$ is described in Section 4.6.
- $g(p)$: the cost of $matched\text{-}vertices(p)$ and $matched\text{-}edges(p)$. Both h 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.

4.3 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.

4.3.1 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 ith vertex of the graph g_1 to the jth 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 $\varepsilon - \varepsilon$.

Similarly, C_e contains all the possible substitutions, deletions and insertions of edges of g_1 and g_2. C_e is constructed in the very same way as C_v.

4.3.2 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 (Riesen, 2009). From now on, we will refer to the set of sorted vertices as *sorted*-V_1.

4.4 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 siblings. 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) = ϕ, *matched-edges*(r) = ϕ, *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*; 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 siblings *siblings(r)*. Then, *siblings(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*$(lb(p))$ among *siblings(r)* and so on. We backtrack to continue the search for a good edit path by revoking p_{min} (if p_{min} equals *null*) and trying out the next child in the set of *siblings(r)* and so on.

4.5 Reduction Strategy

As in *A**, pruning, or bounding, is achieved thanks to $h(p)$, $g(p)$ and a global upper bound UB obtained at node leaves. Formally, for a node p in the search tree, the sum $g(p) + h(p)$ is taken into account and compared with UB. That is, if $g(p) + h(p)$ is less than UB then p can be explored. Otherwise, the encountered p will be pruned from *OPEN* and a backtracking is done looking for the next promising node and so on until finding the best UB that represents the optimal solution of *DF-GED*. This algorithms differs from *A** 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.

4.6 Lower Bound

The lower bound *lb(p)*, adapted to *DF-GED*, is the one used in section *A*GED*, see Section 3.1.

4.7 Upper Bound

The very first upper bound is computed by Munkres' algorithm as it provides fast preliminary results on the GED problem, see (Riesen, 2009) for more details. Afterwards and while traversing the search tree, the upper bound *UBCOST* is replaced by the best *UBCOST* found so far (*i.e.,* a complete path whose cost is less than the current *UBCOST*). After finishing the traversal of the search tree (*i.e.,* when p_{min} is empty and its parent is *r*), the best *UBCOST* is outputted as an optimal solution of *DF*. 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**.

4.8 Pseudo Code

As depicted in Algorithm 2, *DF-GED* starts by a preprocessing step (line 2), then an upper bound *UBCOST* is calculated by *BP* (line 3). The traversal of

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 *UBCOST* and a minimum cost edit path (*UB*) from g_1 to g_2 *e.g.*, $\{u_1 \rightarrow v_3,\ u_2 \rightarrow \varepsilon,\ \varepsilon \rightarrow v_2\}$

1: $OPEN \leftarrow \{\phi\}$, $p_{min} \leftarrow \phi$
2: Generate C_v, C_e and *sorted*-V_1
3: $(UB\ ,\ UBCOST) \leftarrow$BP$(g_1,g_2)$
4: **for** $w \in V_2$ **do**
5: $\quad OPEN \leftarrow OPEN \cup \{u_1 \rightarrow w\}$ s.t. $u_1 \in$ *sorted*-V_1
6: **end for**
7: $OPEN \leftarrow OPEN \cup \{u_1 \rightarrow \varepsilon\}$
8: $r \leftarrow$ parent(u_1), $parent_{tmp} \leftarrow r$
9: **while** true **do**
10: $\quad p_{min} \leftarrow$ bestSibling$(parent_{tmp})$
11: $\quad$**while** p_{min} is not empty and $parent_{tmp}$ does not equal to r **do**
12: $\quad\quad parent_{tmp} \leftarrow$ backtrack$(parent_{tmp})$
13: $\quad\quad p_{min} \leftarrow$ bestSibling$(parent_{tmp})$
14: $\quad$**end while**
15: $\quad$**if** p_{min} is empty and $parent_{tmp}$ equals to r **then**
16: $\quad\quad$Return *UB* and *UBCOST*
17: $\quad$**end if**
18: $\quad OPEN \leftarrow OPEN \setminus p_{min}$
19: $\quad$**if** $g(p_{min}) + h(p_{min}) < UB$ **then**
20: $\quad\quad$**if** *pending-vertices*$_1(p_{min})$ is not empty **then**
21: $\quad\quad\quad$**for** $w \in$ *pending-vertices*$_2(p_{min})$ **do**
22: $\quad\quad\quad\quad p \leftarrow p_{min} \cup \{u_{k+1} \rightarrow w\}$
23: $\quad\quad\quad\quad$**if** $g(p) + h(p) <UBCOST$ **then**
24: $\quad\quad\quad\quad\quad OPEN \leftarrow OPEN \cup \{p\}$
25: $\quad\quad\quad\quad$**end if**
26: $\quad\quad\quad$**end for**
27: $\quad\quad\quad p \leftarrow p_{min} \cup \{u_{k+1} \rightarrow \varepsilon\}$
28: $\quad\quad\quad$**if** $g(p) + h(p) <UBCOST$ **then**
29: $\quad\quad\quad\quad OPEN \leftarrow OPEN \cup \{p\}$
30: $\quad\quad\quad$**end if**
31: $\quad\quad$**else**
32: $\quad\quad\quad$Generate a complete solution $p \leftarrow p_{min} \cup \bigcup_{w \in pending-vertices_2(p)} \{\varepsilon \rightarrow w\}$
33: $\quad\quad\quad$**if** $g(p) + h(p) <UBCOST$ **then**
34: $\quad\quad\quad\quad UBCOST \leftarrow g(p)$
35: $\quad\quad\quad\quad UB \leftarrow p$
36: $\quad\quad\quad$**end if**
37: $\quad\quad$**end if**
38: $\quad$**end if**
39: $\quad parent_{tmp} \leftarrow p_{min}$
40: **end while**

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 to 7). A branching step is performed in line 10 where the best sibling p_{min} is selected, the backtracking is done when there is no more siblings to explore in the selected branch and the parent node ($parent_{tmp}$) is not the root (lines 11 to 14). 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 20 to 30). Similar to *A*GED*, if *pending-vertices*$_1(p_{min})$ is empty, *pending-vertices*$_2(p_{min})$ will be inserted (line 32). *UB* and *UBCOST* are updated whenever a better *UBCOST* is encountered (lines 34 and 35). Each time the next parent to be explored will be replaced by p_{min} (line39). 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.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

4.9 Deadlocks to be Released

DF-GED ends up finding the optimal edit path between two graphs. However, its bottleneck comes from its CPU consumption. Indeed, *DF-GED* is computationally complex as the size of the search space increases exponentially with the number of nodes of the involved graph. Such facts restrict it to work on small graphs only. On the other hand, approximate methods often have a polynomial running time in the size of the input graphs which is much faster, but are not guaranteed to find the optimal solution. We, authors, believe that the more *complex* the graphs, the larger the error committed by the approximate methods. The graphs are more complex when they have more vertices and a well-connected structure. For all these reasons we shed light on the necessity of having a distributed *DF-GED* in order to work on larger graphs that cannot be matched using *DF-GED*. Scaling-out is essential and flexible as one can add *M* machines for enhancing the execution time.

5 OUR DISTRIBUTED PROPOSAL

Recently, people have been increasingly flooding onto deploying their applications on servers which respond to more complicated requirements: *scalability, productivity* and *performance*. To respond to this demand, an application server is proposed in this research. We present a distributed *DF-GED* algorithm, denoted by *D-DF*. This proposed algorithm is built

on top of *Hadoop-MapReduce*. The search tree is decomposed into smaller trees, or sub-trees and the exploration of the partial edit paths is done in parallel.

Definition 2. Scalability
The capability of a system or a process to handle a growing amount of work, *e.g.,* larger graphs, or to be easily expanded or upgraded, when required, to accommodate that amount of work. This is done by adding more machines to the distributed system.

For the sake of programming ease, we have chosen *MapReduce* as a model on which we build our distributed approach. *MapReduce* is easier to program, even for programmers who have no much background about parallel computing as it hides the details of parallelization, fault-tolerance, data distribution and load balancing from them.

5.1 MapReduce Framework

MapReduce is a GOOGLE parallel computing framework (Dean, 2004) used to process vast amounts of data in-parallel. In the MapReduce model, the work is divided into two phases: a Map phase and a Reduce phase. Both of these phases work on key-value pairs. These pairs are defined by the MapReduce programmers depending on the problems they are handling (*e.g.,* words in a text paired by their number of occurrences).

DF-GED is a data-intensive task where lots of edit paths are extracted searching for a best complete edit path to transform g_1 into g_2. MapReduce is a Single Program Multiple data (SPMD) model, dedicated to data-intensive tasks. As a consequence, such a parallel computing model suits *DF-GED*, thanks to its scalability and its ability to be run on a large cluster of machines.

Definition 3. Job
A MapReduce job contains a Map procedure and a Reduce procedure. Each procedure has one or more workers (*i.e.,* Map or Reduce workers that are assigned to Map or Reduce tasks, respectively). Map and Reduce tasks are run by their workers in parallel.

MapReduce uses file-systems as a communication way between Map and Reduce functions. In other words, MapReduce programs read and write theirs results to disks. Figure 3 represents a MapReduce job. In MapReduce, the input files are split into N pieces, or splits, by the MapReduce library in the user program. Then the master, which has a special copy of the program, assigns Map and Reduce tasks to Map workers and Reduce workers, respectively. Each Map worker is assigned one or more file splits (*i.e.,* one or more map tasks), depending on its idleness. When a Map worker finishes its associated task, it saves its buffered key-value pairs to the local disk. The local locations of these buffers are sent back to the master which will inform Reduce workers about them when all the Map workers finish their assigned tasks. Reduce workers read these buffers from their locations, group data by their keys and start to execute the Reduce function.

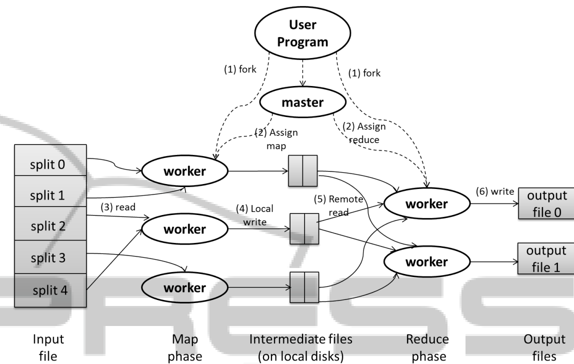

Figure 3: MapReduce Job (Dean, 2004).

5.2 Architecture

The time complexity of *DF-GED* is exponential in the number of vertices of the involved graphs (*i.e.,* g_1 and g_2), in order to decrease the execution time of *DF-GED*, a search tree decomposition is proposed here to fragment the graph matching problem into smaller problems that are solved in a parallel manner. We refer to this distributed approach as *D-DF*. *D-DF* has a single job that has a Map phase without a Reduce one. It starts with two graphs g_1 and g_2 and ends up finding the optimum distance (d).

Figure 4 demonstrates *D-DF's* architecture. *D-DF* is divided into two phases:

- Initialization phase: a certain number of edit paths is generated using *A*-GED*. Moreover, a first upper bound (*UB-best*) is computed using a bipartite graph matching (*BP*). The interested reader is referred to (Riesen, 2009) for more details.
- Distributed phase: each worker on the map side takes an edit path to be solved using *DF-GED*. When a map worker m_i succeeds in finding a better upper bound, it updates the value of *UB-best* and notifies the other map workers so that they get the new value. Also, when a worker is done with one edit path, it communicates with the master program in order to get another edit path that has not been explored yet. This job finishes when there is no more edipath to explore. The value *UB-best* represents the optimal solution (*i.e.,* distance) between g_1 and g_2.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

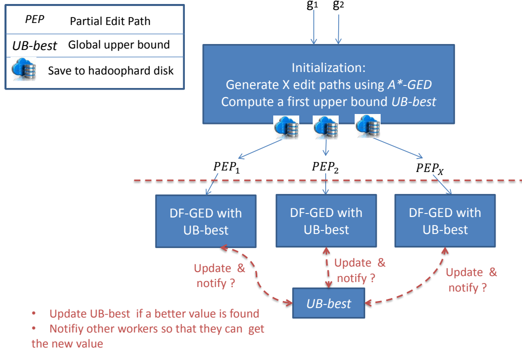

Figure 4: The architecture of *DF-F*.

6 EXPERIMENTS

6.1 Environment

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

6.2 Protocol and Quality Measures

In this section, we explain the protocol used to evaluate the two sequential and optimal approaches (*A*GED* and *DF-GED*). This Protocol is three-fold:

- Calculating the *distance matrix* under a small time constraint.
- Calculating the *distance matrix* under a big time constraint.
- 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 different methods as well as their speed under a time constraint C_T and a memory constraint C_M when graphs sizes increase (*i.e.*, on GREC*k*) and also on *GREC-mix*. To this objective, we test the accuracy of $\mathcal{P}$ when C_T is small (C_T= 350ms) and when C_T is big (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:

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

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 millisecond 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.*, $g(p); h(p)$; 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.

7 RESULTS AND DISCUSSION

Figure 5 shows the deviation results under 350 milliseconds 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 6 demonstrates the running time of both *A*GED* and *DF-GED*. When C_T = 350 milliseconds 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 4.3. On the other hand,

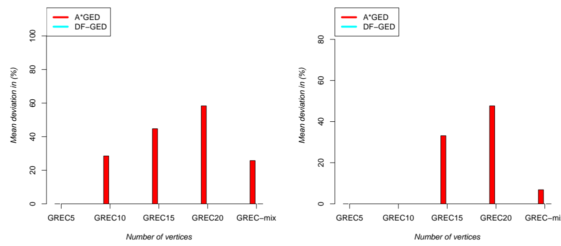

Figure 5: Deviation. Left:(350 milliseconds), Right:(5 minutes).

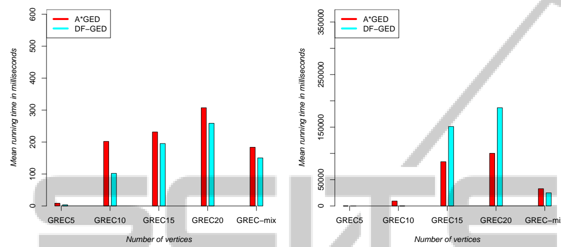

Figure 6: Running Time. Left:(350 milliseconds), Right:(5 minutes).

*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.

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 milliseconds. Moreover, *DF-GED* is 1.7 times faster as the average time *t* is smaller.

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

Algorithms	*t*	*AC*
*A*GED*	119491.5 ms	42.23%
DF-GED	69006.3 ms	98.48 %

8 CONCLUSION AND PERSPECTIVES

As presented in this report, we have discussed the first part of the thesis, 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 cannot match large graphs due to the exponential complexity of GED computation. In this report, we proposed another exact GED algorithm, *DF-GED*, which is based on a depth-first tree 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 accuracy, speed and classification rate. For future work, we aim at measuring the quality of solutions found by our distributed approach (*D-DF*) as well as the aforementioned approximate methods. All these different graph edit distance computations will be evaluated on different PR databases. We expect *D-DF* to outperform other methods in terms of accuracy. We also want to conduct a scalability study to show the importance of such a flexible distributed approach for solving PR problems.

In the next phase of the thesis, we also aim at proposing a distributed *graph-indexing* approach where a graph *g* is fragmented and indexed in a fully distributed manner.

REFERENCES

(1998). A genetic algorithm and its parallelization for graph matching with similarity measures. 2(2):68–73.

A.D.J. Cross, R. W. and Hancock, E. (1997). Inexact graph matching using genetic search. *Pattern Recognition*, pages 953–970.

Allen, R., C. L. M. S. T. S. S. L. and Yasuda, D. (1997). A parallel algorithm for graph matching and its maspar implementation. *Pattern Recognition*, page 490501.

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

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

Bunke, H. (1983). Inexact graph matching for structural pattern recognition. *Pattern Recognition Letters*, 1(4):245–253.

Combier, C., Damiand, G., and C., S. (2013). Map edit distance vs graph edit distance for matching images. In *Proc. of 9th Workshop on Graph-Based Representation in Pattern Recognition (GBR)*, volume 7877, pages 152–161.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Conte, D., Foggia, P., Sansone, C., and Vento, M. (2004). Thirty Years of Graph Matching. 18(3):265–298.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition.

Dean, J., G. S. (2004). Mapreduce : Simplified data processing on large clusters. *Symposium on Operating Systems Design and Implementation.*, 28:137149.

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

Finch, Wilson, e. a. (1998). An energy function and continuous edit process for graph matching. *Neural Computat*, 10.

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.

Kollias, G. (2012). Fast parallel algorithms for graph similarity and matching.

Kuner, P. and Ueberreiter, B. (1988). Pattern recognition by graph matching: Combinatorial versus continuous optimization. *International Journal in Pattern Recognition and Artificial Intelligence*, 2:527542.

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.*, 28:163172.

Patwary, M. M. A., Bisseling, R. H., and Manne, F. (2010). Parallel greedy graph matching using an edge partitioning approach. *Proceedings of the fourth international workshop on High-level parallel programming and applications - HLPP '10*, page 45.

Plantenga, T. (2013). Inexact subgraph isomorphism in mapreduce. *Journal of Parallel and Distributed Computing*, page 164175.

Qiu, H. and Hancock, E. R. (2006). Graph matching and clustering using spectral partitions. *Pattern Recognition*, 39(1):22–34.

Riesen, K., B. H. (2009). Approximate graph edit distance computation by means of bipartite graph matching. *Image and Vision Computing.*, 28:950959.

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

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

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). *IEEE Transactions on Systems, Man and Cybernetics*, pages 48–62.

Vento, M. (2014). A long trip in the charming world of graphs for pattern recognition. *Pattern Recognition*.

W. Christmas, J. K. and Petrou., M. (1995). Structural matching in computer vision using probabilistic relaxation. *IEEE Trans. PAMI,*, 2:749764.

Zeng, Z., Tung, A. K. H., Wang, J., Feng, J., and Zhou, L. (2009). Comparing stars: On approximating graph edit distance.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS