Pattern Recognition Letters 84 (2016) 215–224

Contents lists available at ScienceDirect

Pattern Recognition Letters

journal homepage: www.elsevier.com/locate/patrec

Anytime graph matching

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

Laboratoire Informatique, Université François Rabelais de Tours, 64 avenue Jean Portalis, 37200 Tours, France

ARTICLE INFO

Article history:
Received 30 November 2015
Available online 15 October 2016

MSC:
41A05
41A10
65D05
65D17

Keywords:
Graph matching
Anytime algorithms
Pattern recognition
Graph edit distance
Performance evaluation metrics

ABSTRACT

In this paper, we propose and explain the use of anytime algorithms in graph matching (GM). GM methods have been involved in many pattern recognition problems. In such a context, GM methods are part of a more complex retrieval system that imposes time and memory constraints on such methods. Anytime algorithms are well suited for use in such an uncertain environment. An anytime algorithm quickly provides the first solution to the problem, finds a list of improved solutions and eventually converges to the optimal solution instead of providing one and only one solution (i.e., the optimal solution). We describe how to convert a recent depth-first GM method into an anytime one. By constraining the solver, the algorithm creates an anytime heuristic search algorithm that allows a flexible trade-off between the search time and the solution quality. We analyze the properties of the resulting anytime algorithm and consider its performance in terms of the deviation of the provided solution from the optimal or the best one found by a state-of-the-art method. Experiments were carried out on seven different types of graph datasets. Moreover, the adopted algorithm was compared to four approximate error-tolerant GM methods. Results showed that the anytime GM can outperform suboptimal methods by only waiting for a small amount of supplementary time. This conclusion brings into question the usual evidence that claims that it is impossible to use optimal GM methods in real-world applications.

1. Introduction

Powerful data structures, such as attributed graphs, that are used to represent complex entities always require more and more computational resources. Thus, a trade-off between accuracy and computational cost (i.e., execution time and consumed memory) has to be found. On this basis, converting algorithms into anytime algorithms is of great benefit [10,35]. The main idea behind anytime algorithms comes from the simple observation that there is no reason to stop an algorithm after the first solution is found, especially when it is possible to find a better solution with plenty of available time. By continuing the search, the algorithm can find a sequence of improved solutions and eventually with additional time, it can even converge to an optimal solution.

Speaking of powerful data structures, attributed graphs have become more and more popular in many different fields, e.g., datamining and pattern recognition. In this context, efficient error-tolerant GM methods are of high interest. Error-tolerant GM methods can provide precise correspondences between the vertices and the edges of two graphs. In the literature, many different GM algorithms have been proposed [5,31]. However, the complexity of exact GM methods is NP-hard. Such a fact restricts their applicability to graphs with a rather small size.

At present, two main families of error-tolerant GM methods can be found in the literature: exact and approximate. Few exact methods have been found in the literature [2,12,24]. On the other hand, a number of approximate GM methods have been proposed with reduced computational time and accuracy. Some of these methods reduce the flexibility to work on graphs with different structures and attributes. Among these methods, we mention the spectral methods [30], and the methods that are restricted to planar graphs [11] and trees [29], to name a few of them. Some approximate methods also work directly on the adjacency matrix of the graphs relaxing the combinatory optimization to a continuous one e.g., path following in [32]. The graduated non-convexity and graduated concavity procedure (GNCGCP) was proposed by Liu and Qiao [17] as a general optimization framework to suboptimally solve the combinatorial optimization problems such as error-tolerant GM. Other approaches can also be found in the literature such as tree-based methods [19] and linear sum assignment solver (e.g., bipartite GM (*BP* in [20] and Square Fast *BP* in [27]).

In this work, we would like to take advantage of these two aforementioned types of GM methods by merging them together to propose a third type of GM methods that we call "*Anytime GM*". On this basis, GM methods can be categorized differently. The first, are methods that are fast (enough) but that can only find one

* Corresponding author.
E-mail address: zeina.abu-aisheh@univ-tours.fr (Z. Abu-Aisheh).

http://dx.doi.org/10.1016/j.patrec.2016.10.004
0167-8655/© 2016 Elsevier B.V. All rights reserved.

feasible solution (e.g., [20,27]. The second, are methods that are tree-search based (e.g., [2,12,19] that can provide more than one solution while traversing the search tree during the matching process. Tree-based methods have become of great interest since computational time and even the explored search space can be manageable with the impact of the quality of the provided matching solution. From here comes the primary motivation of the paper which says that tree-based methods for GM computation can be turned into anytime methods by varying the computational time and studying the effect on the outputted answers. In this paper, we propose an anytime GM algorithm based on a depth-first GM algorithm [2]. This algorithm does not consume so much memory. By managing time and memory at the same time, the proposed method becomes as scalable as possible. Another contribution of the paper is the experimental protocol where information is provided about GM quality while increasing time constraints.

The rest of the paper is organized as follows. In Section 2, we introduce the problem statements of GM and the anytime algorithms. In Section 3, we review related work by describing the main works on exact and approximate error-tolerant GM. We also discuss the background of anytime methods and show the interest of making GM methods anytime compliant. In Section 4, we present our anytime version of GM computation. In Section 5, this method is compared to approximate ones using adequate GM evaluation metrics in [1] that evaluate both precision and run time. Finally, Section 6 offers some conclusions and suggestions for future work.

2. Problem statement

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 $V_1 = (u_1, \ldots, u_n)$ and $V_2 = (v_1, \ldots, v_m)$ the sets of vertices of G_1 and G_2, respectively. E_1 and E_2 represent the edges of G_1 and G_2, successively, whereas the terms μ and ζ refer to the attributes on vertices and edges, respectively. In error-tolerant GM, a measurement of the strength of matching vertices and/or edges of two graphs G_1 and G_2, referred to as penalty cost, is applicable on both graph structures and attributes. The basic idea is to assign a penalty cost to each matching operation according to the amount of distortion that it introduces in the transformation. A set of operations that transforms G_1 into G_2 is called *Edit Path* in the literature [21]. When (sub)graphs differ in their attributes or structures, a high penalty cost is added during the matching process. Such a cost prevents dissimilar (sub)graphs from being matched since they are different. Likewise, when (sub)graphs are similar, a small penalty cost is added to the overall cost. This cost includes matching two vertices and/or edges, inserting a vertex/edge or deleting a vertex/edge. The question of finding the minimum cost matching is a discrete optimization problem. Error-tolerant GM is NP-hard and thus algorithms that solve optimally error-tolerant GM suffer from both memory and time consumption. On this basis, researchers have shed light on the approximate methods that can find suboptimal solutions that are hopefully close to the optimal ones; however, the quality of the solutions in function of the solving time has not been deeply studied yet.

In this paper, we establish a compromise between exact and approximate error-tolerant GM algorithms, referred to here as anytime algorithms.

The concept of anytime algorithms was first reported in [36]. The desirable properties of anytime algorithms are as follows:

- Interruptibility: After some small amount of setup time,[1] a suboptimal solution can be provided by stopping the algorithm at time *t*.

[1] The time needed to output a first solution by an anytime method.

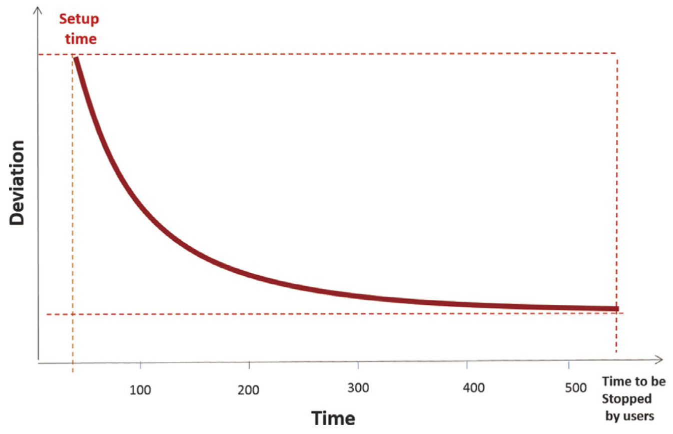

Fig. 1. Characteristics of anytime algorithms.

- Monotonicity: The quality of the result increases as a function of computational time.
- Measurable quality: We can always measure the quality of a suboptimal result.
- Diminishing returns: At the beginning of anytime algorithms, the improvement in the solutions can be remarkably observed. However, this improvement decreases over time.
- Preemptability: Anytime algorithms can be suspended and resumed with minimal overhead.

Anytime algorithms have a trade-off between quality and execution time, see Fig. 1. They can find the first best-so-far solution after some setup time at the beginning of the execution. From Fig. 1, one can see that the quality of the solution improves with increasing execution time. Users have the choice of stopping the algorithm at anytime and thus getting an answer that is satisfactory, or they can run their algorithm until its completion when it is important to find the optimal solution. It is hard to know when an anytime algorithm should be interrupted (by the system or the user) to get the best-so-far answer. Thus, algorithms should be equipped with the appropriate stopping criteria based on the monitoring of the actual performances when the time of an optimal interruption is not known in advance.

The setup time needed by anytime algorithms is a crucial point for several reasons. First, to be able to quickly provide a solution and then to be stopped by the user. Second, to be able to provide a specified response time. For any kind of graphs, users are sure that the matching will take no longer than the specified time. Third, to not let users wait specially when having a reactive system. A study of this specific point will be proposed in the experiments.

3. Related work

This section is organized as follows. First, we shed light on the state-of-the-art of GM methods. Second, the literature of anytime methods is presented aiming at proposing a first anytime GM method. Last, but not least, we show the interest of applying anytime properties in GM methods.

3.1. Graph matching algorithms

3.1.1. Exact error-tolerant graph matching approaches

The A^*-based algorithm is considered as a foundation work for solving GM [24]. The computations are achieved by means of an ordered tree. Such a search tree is constructed dynamically at run time by iteratively creating successor vertices. Only leaf vertices correspond to feasible solutions and, thus, complete matching operations. For a tree node *p* representing a partial matching in the search tree, *g(p)* represents the cost of the partial matching operations accumulated so far, and *h(p)* denotes the estimated costs from *p* to a leaf node representing a complete solution. The sum *g(p)+h(p)* is the total cost assigned to a tree node in the search tree. If *h(p)* is lower or equal than the real costs then *h(p)* is said to be admissible and A^* is guaranteed to found an optimal path from the root node to a leaf node. In the worst case, the space complexity can be expressed as $O(|\gamma|)$ [6] where $|\gamma|$ is the cardinality of the set of all possible edit paths. Since A^* is exponential in the number of vertices involved in the graphs, the memory usage is still an issue.

To overcome the memory problem of A^*, [2] proposed a recent depth-first branch-and-bound GM algorithm, called *DF*. This algorithm speeds up the computations of GM thanks to its upper and lower bounds pruning strategy and its preprocessing step. Moreover, *DF* does not exhaust memory as the number of pending partial solutions that are stored in the set, called *OPEN*, is relatively small thanks to the *DFS* algorithm where the number of pending nodes is $|V_1|.|V_2|$ in the worst case. In both A^* and *DF*, the problem of solving *h(p)* is of first interest. One can map the unprocessed vertices and edges of graph G_1 to the unprocessed vertices and edges of graph G_2 such that the resulting costs are minimal. This mapping should be done in a faster way than the exact computation and should return a good approximation of the true future cost. In Section 4.4, $h(p)$ will be detailed.

To the best of our knowledge, Almohamad and Duffuaa in [3] proposed the first linear programming formulation of the weighted graph matching problem. It consists in determining the permutation matrix minimizing the L_1 norm of the difference between adjacency matrix of the input graph and the permuted adjacency matrix of the target one. More recently, Justice and Hero [12] also proposed a binary linear programming formulation of the graph edit distance problem. GM is treated as finding a subgraph of a larger graph known as the edit grid. The edit grid only needs to have as many vertices as the sum of the total number of vertices in the graphs being compared. One drawback of this method is that it does not take into account attributes on edges which limits the range of application.

3.1.2. Approximate error-tolerant graph matching approaches

The main reason that motivated researchers to solve approximately the problem of error-tolerant GM comes from the combinatorial explosion of the exact error-tolerant approaches. Numerous variants have been proposed for a faster but suboptimal computation of GM. One of the most well-known modifications of A^*, called beam-search (*BS*), has been proposed in [19]. The purpose of *BS* is to prune the search tree while searching for an optimal edit path. Instead of exploring all edit paths in the search tree, the x most promising partial edit paths are kept in the set of promising candidates *OPEN*.

In [20], the GM problem is reduced to a linear sum assignment problem which can be solved in $O(n^3)$ where n is equal to $|V_1|+|V_2|$. A cost matrix is involved in the process to gather vertex-to-vertex costs.[2] In the rest of the paper, this algorithm is referred to as *BP*. Recently, a new version of *BP* for computing GM, called fast bipartite method (*FBP*), has been published in [27]. Such an algorithm obtains the same distance with lower computation time as it reduces the size of the cost matrix. Since *BP* and thus *FBP* consider local structures rather than global ones, the optimal GM is overestimated. Recently, researchers have observed that *BP*'s overestimation is very often due to a few incorrectly assigned vertices. That is, only a few vertex substitutions from the next step are responsible for the additional (unnecessary) edge operations in the step after, thus resulting in the overestimation of the optimal edit distance. In [23], *BP* was used as an initial step. Then, pairwise swapping of vertices (local search) was done aimed at improving the accuracy of the distance obtained so far. In [25], a search procedure based on a genetic algorithm was proposed to improve the accuracy of *BP*. In [8], a beam-search version of *BP* was proposed. This work focuses on investigating the influence of the order in which the assignments were explored. These improvements increase run times. However, they improve the accuracy of the *BP*'s solution.

[2] It also partially integrates edge costs.

3.2. Anytime tree-search based algorithms

Tree-search based GM algorithms are considered as anytime algorithms since they can find several solutions while exploring their search space. Thus, in this section, these algorithms will be surveyed aiming at proposing an anytime GM method.

3.2.1. Time bottleneck and anytime algorithms

The most common approach to transform a search algorithm, such as A^*, into an anytime algorithm consists of the following three changes [10].

- A non-admissible evaluation function, $lb_0(p) = g(p) + h_0(p)$, where the heuristic $h_0(p)$ is not admissible, is used to select the nodes for expansion in an order that allows good, but possibly suboptimal, solutions to be found quickly.
- The search continues after a solution is found, to find improved solutions.
- An admissible evaluation function (i.e., a lower-bound function), $lb(p) = g(p) + h(p)$, where $h(p)$ is admissible, is used together with an upper bound (UB) on the optimal solution cost given by the cost of the best solution found so far, to prune the search space and detect convergence to an optimal solution.

On the basis of this idea, many researchers have explored the effect of weighting the terms $g(p)$ and $h(p)$ in the node evaluation function differently, to allow A^* to find a bounded-optimal solution with less computational effort. In the approach called Weighted A^* (WA^*) [16], the node evaluation function is defined as $lb_0(p) = g(p) + \omega^* h(p)$, where the weight ω is a parameter set by the user. If ω is greater than 1.0, the search will not be admissible and the first solution found may not be optimal, although it is usually found much faster. The weighted heuristic accelerates the search for a solution because it makes tree nodes closer to a goal seem more attractive, giving the search a more depth-first aspect and implicitly adjusting a trade-off between search effort and solution quality. The weighted heuristic search is more effective for search problems with close-to-optimal solutions, and can often find a close-to-optimal solution in a small fraction of the time it takes to find an optimal solution. Some variations of the weighted heuristic search have been studied. For example, an approach called dynamic weighting adjusts the weight with the depth of the search [13]. Moreover, a learning real-time A^* ($LRTA^*$) was proposed in [28].

3.2.2. Memory bottleneck and anytime algorithms

The scalability of A^* is limited by the memory required to store the lists of open path inside the search tree. Such a fact limits the scalability of anytime A^*. In the conception of our new GM algorithms, we have to take care of this point and try to create a linear-space anytime algorithm.

Considering the memory aspect, depth-first search (*DFS*) algorithms are very effective for some tree-search problems since they overcome the memory bottelneck from which A^* methods suffer. *DFS* algorithms are anytime by nature [33], as they systematically explore the leaf nodes of a state space. They quickly find a solution that is suboptimal, and then continue to search for an improved solution until an optimal solution is found. They can even use the cost of the best solution found so far as an upper bound to prune the search space. Therefore, the *DFS* strategy seems to correspond to a simple and efficient approach for converting an optimal GM algorithm into an anytime one that offers a trade-off between search time, memory consumption and quality of the provided solution when more time is available.

Several variants of A^* have been developed that use less memory, including algorithms that require only linear space in the depth of the search space. One of the most known algorithms is recursive best-first search (*RBFS*) in [14]. *RBFS* is a weighted heuristic search algorithm that expands frontier nodes in best-first order. It saves memory by determining the next node to expand using stack-based backtracking instead of selecting nodes from an open list that contains the search tree nodes to be processed.

3.3. The interest of anytime algorithms in graph matching

From the aforementioned sections, we can conclude that only a few exact GM approaches have been proposed to postpone the graph size restriction [2,12,24]. Some approximate GM methods (e.g., [15,20,27,32] have a polynomial running time in the size of the involved graphs and thus are much faster than the optimal ones. In these types of algorithms, increasing the time will not lead to the improvement in the quality of the found solution. Moreover, the more complex the graphs, the larger the error committed by these methods. Graphs are generally more complex in cases where neighborhoods and attributes do not allow to easily differentiate between vertices. On the other hand, the behavior of the iterative algorithms (e.g., [8,23,25] is similar to the anytime ones. These algorithms can be converted to anytime algorithms because they can find several solutions during the matching process. However, there are no studies of the quality of the outputted solutions as a function of time.

In this paper, we propose to define a third category for anytime GM methods that will allow a trade-off between the valuable properties of both the previously existing types of GM methods: speed for suboptimal methods and quality of the provided solution for optimal ones. We believe that such anytime GM methods are of great interest. They can output the first solution (if the solution is satisfactory enough) or they can explore the search space in order to improve the solution (when given more time). In the rest of the paper, we shall demonstrate the benefit of anytime GM methods.

4. Proposed anytime graph matching algorithm

This section describes how we convert the arbitrary GM problem into an anytime one. The algorithms that are dedicated to solving the GM problem can produce an instant matching between two graphs. If they are given the luxury of additional time, they can increase the precision of this matching. Anytime algorithms find the first solution and continue the search to improve it. Each time a new solution is found, it is saved (or outputted). Our algorithm, referred to as anytime depth-first (*ADF*), is an adapted version of the *DF* algorithm in [2] in which important properties for anytime algorithms are added and studied such as interruptibility, monotonicity and measurable quality, see Section 2. The following sections describe the main parts of this algorithm in detail.

4.1. Pre-processing

Before starting the branch-and-bound part, *DF* initializes the important data structures to speed up the tree search exploration. Preprocessing includes two steps: cost matrices construction and vertex-sorting strategy.

4.1.1. Cost matrices

The vertex and edges cost matrices (C_v and C_e) are constructed, respectively. This step aims to speed up the branch-and-bound part by getting rid of the re-calculations of the assigned costs when matching the vertices and edges of G_1 and G_2.

A vertex cost matrix C_v, whose dimension is $(n+2)\times(m+2)$, is constructed as follows:

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

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 graph G_1 to the jth vertex of graph G_2. The left upper corner of the matrix contains all possible vertex substitutions, whereas the right upper corner represents the cost of all possible insertions and deletions of the 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 the edges of G_1 and G_2. C_e is constructed in the very same way as C_v.

4.1.2. Vertex-sorting strategy

To speed up the exploration of the search tree while searching for the optimal GM, it is important to sort V_1 to start with the most promising vertices. To sort V_1, the algorithm applies *BP* [20] to obtain a suboptimal edit path ($EP = \{u_i \rightarrow v_k, \cdots, u_n \rightarrow v_l, \cdots\}$ with $u \in V_1$ and $v \in V_2$). From this edit path, vertex-to-vertex mapping costs are used to sort V_1 in ascending order. *BP* [20] outputs an initial edit path *EP* and its distance d_{BP} which can then be used as a first *UB*. Then V_1 is sorted according to the matching weight C_{ij} of the cost matrix *C*. That is, each u_i is given a weight that corresponds to the matching cost of $u_i \rightarrow v_{ik} \in EP$.

4.2. Branch-and-bound

4.2.1. Tree node structure

Each tree node p in the search tree contains information about the matched vertices and edges of G_1 and G_2 in p. It also contains, the estimated future cost, referred to as $h(p)$ [24], from node p which does not overestimate the cost of the complete solution. This function is described in Section 3.1.1. In addition to $h(p)$, the total cost of the matched vertices and edges, referred to as *g(p)*, is also included in each node p. Both h and g depend on the attributes as well as on the structure of the involved sub-trees. The cost functions involved with each dataset permit to calculate the insertions, deletions and substitutions of vertices and/or edges.

4.2.2. Branching and selection strategies

The solution space is organized as an ordered tree which is explored in a depth-first way. In DFS, 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 exploration starts with the root node. In order to generate the children of tree nodes, each tree node p takes the next most promising vertex u_i in the sorted list of V_1 and generates some edit paths by matching u_i with all the non-matched vertices of G_2 in addition to deleting u_i (i.e., $u_i \rightarrow \epsilon$). Afterwards, the children of p are sorted in an ascending manner according to *lb(q)*. Then these children are added to *OPEN*. Since the children are sorted in ascending order, the exploration is achieved by choosing the first element in *OPEN* to be explored and so on. Thus, each node is visited just before its children.

4.3. Reduction strategy

As in A^*, pruning, or bounding, is achieved thanks to *h(p)*, *g(p)* and a global *UB* obtained at the node leaves. Formally, for a node p in the search tree, *lb* 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 the next promising node is evaluated and so on until the best *UB* is found that represents the optimal solution of *ADF* or until the process is interrupted by the timer since it is an anytime algorithm. This algorithms differs from A^* since at anytime t, in the worst case, *OPEN* contains at most $|V_1|.|V_2|$ elements and hence the memory consumption is not exhausted.

4.4. Upper and lower bounds

The estimation of *h(p)* should be done in a faster way than the exact computation and should return a good approximation of the true future cost. In our proposal, *h(p)* is calculated via a bipartite heuristic [24]. This is achieved by mapping vertices the unprocessed vertices and edges of graph G_1 to the unprocessed vertices and edges of graph G_2 such that the resulting costs are minimal. On the basis of the cost matrices C_v and C_e, Munkres' algorithm [18] can be executed separately on vertices and edges. This algorithm finds the optimal, i.e., the minimum cost, assignment of the elements (vertices or edges) represented by the rows to the elements represented by the columns of matrix C_v or C_e in polynomial time. That is, in the worst case the maximum number of operations needed is $\mathcal{O}((n+m)^3)$, where $(n+m)$ is the dimensionality of the cost matrix. 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 *OPEN* equals $\{\phi\}$), the algorithm outputs the best *UB* as an optimal solution of *ADF*. Encountering upper bounds when performing a depth-first traversal efficiently prunes the search space and thus helps in finding the optimal solution faster than A^* does.

4.5. Anytime properties

The time needed to find the first solution is called *setup time*. One has to decide whether having a long setup time and thus finding a satisfactory first solution for users or taking a shorter setup time and thus finding a less satisfactory first solution. In our algorithm, an initial solution can be computed using *BP* in cubic time or it can remain unset until the first branch is explored in quadratic time. This choice can be seen as a parameter. Other decisions can also be done but they are out of the scope of this paper.

ADF guarantees to find the optimal solution of *GM*(G_1, G_2) if no time limit is set. It also regularly provides better and better solutions and exports all of them while exploring the search tree.

One should also notice that having a sufficiently good first solution can have an important impact on the time needed to find the next better solutions. That is, the setup time and the convergence slope are closely coupled.

Algorithm 1 Anytime depth-first GM algorithm (ADF).

Input: Non-empty attributed graphs $G_1 = (V_1, E_1, \mu_1, \zeta_1))$ and $G_2 = (V_2, E_2, \mu_2, \zeta_2)$ where $V_1 = \{u_1, \ldots, u_{|V_1|}\}$, $V_2 = \{v_1, \ldots, u_{|V2|}\}$, μ and ζ are the attributes associated with the vertices and edges, respectively.

Output accessible at anytime: *UB* = the current minimum edit path cost and *BestEditPath* = sequence of edit operations.

```
1:  Initialization: $OPEN \leftarrow \emptyset$, $BestEditPath \leftarrow \phi$, $UB \leftarrow \infty$
                         Pre-processing:
2:  Generate $C_v$, $C_e$
3:  Optional: {Steps below are optional}
        $(UB, BestEditPath) \leftarrow BP(G_1, G_2)$
        Export($UB$,$BestEditPath$)
        $\bar{V_1} \leftarrow$ Sort($V_1$) {in ascending order of $BP(G_1, G_2)$}
                         Branch-and-Bound:
4:  $root \leftarrow \phi$
    Generate the children of $root$, sort them in ascending order of
    $g + h$ and insert them into $OPEN$
5:  while $OPEN$ != $\emptyset$ do
6:    Take the first element $p_{min}$ and remove it from $OPEN$
7:    Generate the children of $p_{min}$, sort them in ascending order
      of $g + h$ and insert them into $OPEN$
8:    if $p_{min}$ has no children then
9:      Insert all non-matched vertices of $V_2$ into $p_{min}$
10:     if $g(p_{min}) < UB$ then
11:       $UB \leftarrow g(p_{min})$, $BestEditPath \leftarrow p_{min}$
12:       Export($UB$,$BestEditPath$)
13:     end if
14:   end if
15: end while
```

4.6. Pseudo code

As depicted in Algorithm 1, *ADF* starts by the initialization and pre-processing steps (lines 1 to 3). First, C_v and C_e are generated (line 2). Second, *UB* is set to ∞ or calculated by *BP* (line 3). Third, V_1 is sorted according to the distances obtained in the matrix of *BP* (line 3) resulting in a new list, referred to as $\bar{V_1}$. Note that if *BP* is not used as an upper bound, V_1 will not be sorted. The traversal of the search tree starts by generating the root's children (line 4). The most promising vertex u_1 is taken from $\bar{V_1}$. Consequently, vertex u_1 is substituted with all the vertices in V_2. In addition, the deletion of u_1 is also generated (i.e., $u_1 \rightarrow \epsilon$). The children (i.e., mappings between vertices) are sorted in ascending order of $g + h$ and then inserted in *OPEN* (line 4). Since the children inserted in *OPEN* are ordered, the most promising child p_{min} at the deepest level in the search tree will be first selected (line 6). The children of p_{min} are generated by substituting vertex $u_i \in V_1$ with the unmatched vertices in V_2 in addition to its deletion (line 7). On the other hand, if all the vertices of V_1 are matched, all the unmatched vertices of V_2 will be inserted in p_{min} (line 9). *UB* and *BestEditPath* are updated whenever a better solution is encountered (lines 10 to 13). Note that the output is available at anytime after the setup time. As long as there is some available time and there are nodes to explore in *OPEN*, the exploration step continues (line 5). Note that edge operations are taken into account in the matching process when substituting, deleting or inserting their corresponding vertices.

5. Experiments

This section describes the protocol and shows the experimental results that prove the validity of the approach. A

Table 1
Methods included in the experiments.

Acronym	Reference	Details
ADF	This paper	AnyTime GM
BS-1 and *BS-100*	[19]	beam-search with OPEN size = 1 and 100, respectively
BP	[20]	The bipartite GM
FBP	[27]	Fast *BP*
SBP-Beam	[8]	Sorted beam-search *BP* where the sorting strategy is *deviation-inverse*
JHBLP	[12]	A binary linear GM formulation

demonstration of the concept of anytime GM can be found on the following website: http://www.rfai.li.univ-tours.fr/PagesPerso/zabuaisheh/anytimeGM.html.

5.1. Included methods

Table 1 summarizes the methods included in the experiments. The state-of-the-art methods were not anytime methods. However, for the experimental evaluations, we added the time interruption property to each of them. Several versions of *ADF* and *BS* are tested where methods with *LB* refer to the versions where *h(p)* is integrated. In addition, *ADF-UB* indicates that the first upper bound (line 3 in Algorithm 1) is integrated. See Section 4.4 for the description of the lower and upper bounds.

In all the aforementioned methods, memory consumption is not exhausted. The memory complexity of *ADF* and *ADF-UB* algorithms is relatively small thanks to the DFS algorithm where the number of pending nodes is $|V_1|.|V_2|$ in the worst case. A^* could also have been added to the experiments, however, its memory complexity is exponential. Therefore, A^* will not be able to keep exploring the search tree and thus outputting feasible (i.e., complete) solutions before timing out. We also implemented the algorithm of [12] which is then solved via the CPLEX-12 mathematical solver. For all graph comparisons, this method was unable to output feasible solutions in 500 milliseconds (ms) or less. This is due to the setup time needed by the mathematical solver, which takes more time to solve the continuous relaxation before starting the tree search exploration.

5.2. Databases

Seven datasets are integrated in the experiments: (GREC, Mutagenicity, Protein, CMU, PAH and two synthetic datasets). Three of them (i.e., GREC, Mutagenicity and Protein) were taken from the IAM Graph Database Repository [22]. The CMU dataset can be found at the CMU website [4]. These four datasets have been recently included in a new repository, called GDR4GED [1], that aims at evaluating the scalability of GM methods. GDR4GED is annotated with GM ground truth. For more information, visit IAPR-TC15's website.[3] In addition to these datasets, a chemical dataset, called PAH, was taken from GREYC's Chemistry dataset repository.[4] Moreover, a new synthetic dataset was generated for experimental evaluations. This dataset was created using the Erdos-Renyi model [7]. The reason for having chosen such datasets is to have a variety of graph attributes (i.e., numeric and/or symbolic attributes on vertices and/or edges or non-attributed vertices and/or edges) and densities (i.e., high and low density graphs). In addition, the number of vertices in these datasets starts from 20 vertices up to 200 vertices.

Table 2 summarizes the characteristics of all the selected datasets.

[3] https://iapr-tc15.greyc.fr/links.html.
[4] https://brunl01.users.greyc.fr/CHEMISTRY/index.html.

Each dataset has specific edit cost functions. Two non-negative meta parameters are associated to GM: (τ_{vertex} and τ_{edge}) where τ_{vertex} denotes a vertex deletion or insertion costs whereas τ_{edge} denotes an edge deletion or insertion costs. A third meta parameter α is integrated to control whether the edit operation cost on the vertices or on the edges is more important. Table 3 demonstrates the cost functions of each of the included datasets as well as their meta parameters. Note that the synthetic datasets, the parameters were taken from the dataset Letter-Low in IAM [22]. The error-tolerant GM matching is more difficult when there are no attributes on vertices and/or edges or when structures are redundant. For instance, matching the graphs of PAH is difficult since it has completely unattributed graphs. On the other hand, matching the graphs of GREC is easier since it is rich with attributes. Note that in our implemented version of *FBP*, the three restrictions on the edit costs were not included [27].

In the experiments, we selected 10 graphs from each of GREC, MUTA and Protein. These graphs represent the maximum number of vertices that was found on each of the datasets. The graphs can be downloaded from the GDR4GED repository [1]. On this basis, 100 pairwise comparisons were carried out on these datasets. As for CMU, one hundred eleven images in total are publicly available in [4]. Six hundred sixty comparisons were carried out. On PAH, 10 graphs whose size varies from 17 to 24 vertices were also selected and, thus, it also results in 100 comparisons. Two synthetic datasets each of which had 10 graphs of 200 vertices were created using the Erdos-Renyi model [7]. Two density graph families can be found: low density (i.e., 0.1) and high density (i.e., 0.4). These densities refer to the probability of having an edge between two vertices. The purpose of such a database was to see how GM methods behave when having low, or high, density graphs. The meta parameters of the synthetic datasets were taken from the Letter dataset [22].

5.3. Environment

The evaluation were conducted on a computer with a 24-core Intel i5 processor at 2.10GHz and 16 GB of memory. A memory constraint was set to 1GB. The time constraint was varied from 5 ms to 500 ms on all databases.

5.4. Protocol

The objective of the experiments was to study the trade-off between the quality and the time of all the methods so as to investigate the matching accuracy in function of the time. Each comparison was tested under a given time and memory constraints.

To evaluate *ADF*, we chose a deviation metric to compare all the included methods, see [1] for more details about the GM evaluation metrics. We compute the error committed by each method m over the reference distances. For each pair of graphs matched by method m, we provide the following deviation measure:

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

where G is the number of graphs. $d(G_i, G_j)^m$ is the distance obtained when matching G_i and G_j using method m and R_{G_i,G_j} corresponds to the best known solution. For the IAM datasets, we used the ground truth of [1] as a reference. The humans' ground truth was used as a reference for CMU. On the other hand, for PAH, optimal solutions were provided by carrying out the computations using the algorithm in [12].

In the experiments, the average deviation was calculated per dataset where the x-axis represents the time limit t and the y-axis

Table 2
The characteristics of the datasets included in the experiments.

Dataset	GREC-20	MUTA-70	Protein-40	CMU houses	PAH	Synthetic (0.1)	Synthetic (0.4)
Vertex labels	x,y coordinates	Chemical symbol	Type and amino acid sequences	None	None	None	None
Edge labels	Line type	Valence	Type and length	Distance between points	None	None	None
$\overline{vertices}$	20	70	40	30	20.7	200	200
$\overline{edges}$	21.6	73.8	78.3	79	24.4	2013	7941.8
Max vertices	20	70	40	30	28	200	200
Max edges	22	75	95	79	34	2095	8089

Table 3
The cost functions and meta parameters of the datasets.

Dataset	GREC-20	MUTA-70	Protein-40	CMU houses	PAH	Synthetic (0.1)	Synthetic (0.4)
τ_{vertex}	90	11	11	∞	3	0.3	0.3
τ_{edge}	15	1.1	1	–	3	0.5	0.5
α	0.5	0.25	0.75	0.5	0.5	0.75	0.75
Vertex substitution function	Extended euclidean distance	Dirac function	Extended string edit distance	0	0	L2 norm	L2 norm
Edge substitution function	Dirac function	Dirac function	Dirac function	Dirac function	0	Dirac function	Dirac function
Reference of cost functions	[26]	[26]	[26]	[34]	[9]	–	–

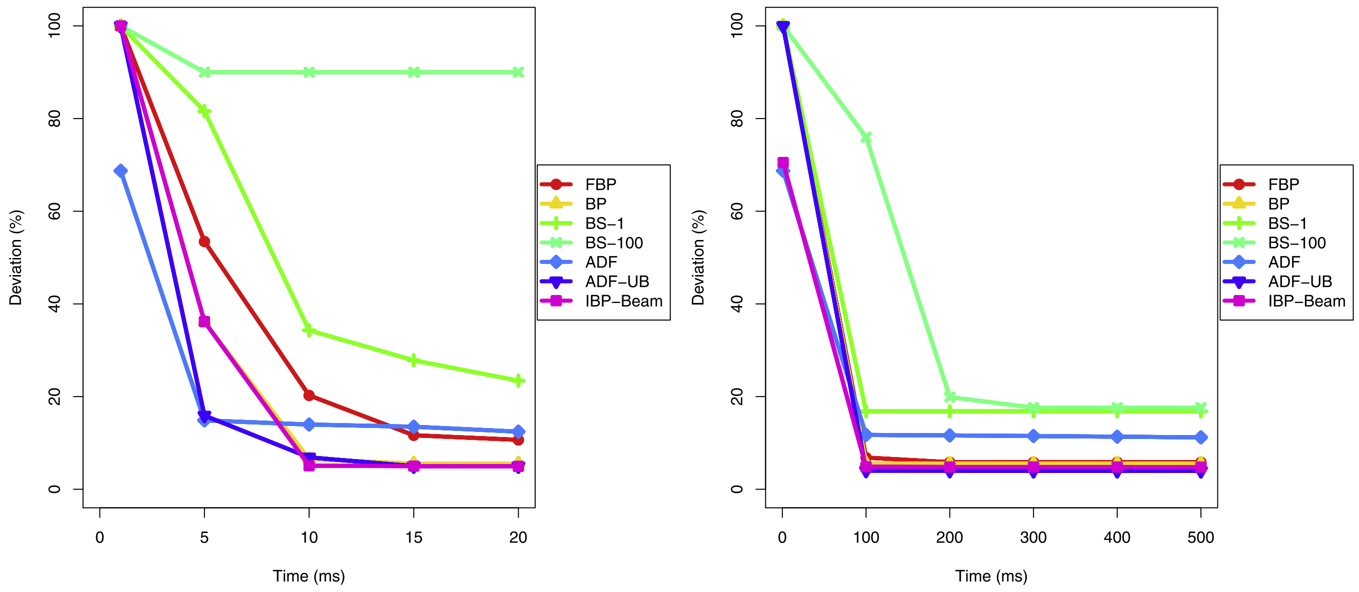

Fig. 2. GREC Deviation: Left (up to 20 ms), Right (up to 500 ms).

shows the average deviation within *t*. If a method *m* did not output a solution before timing out, the deviation would be set to 100%.

We also measured the setup time needed by *ADF* to output an initial solution (i.e., the first complete solution found when exploring the search tree). Only *ADF*, *ADF-UB*, *BS-100* and *SBP-Beam* were able to find one or more solutions while exploring the search tree. This time was compared with the time taken by *BS-1*, *BP* and *FBP*, which outputted one and only one complete solution. Hereafter, this measured time will be called "*setup time*".

6. Results and discussions

Fig. 2 illustrates the deviation on GREC-20 when varying the available time up to 20 ms and 500 ms (see Fig. 2). One can observe that *ADF* was the fastest method to output solutions, followed by *BP*. however, *ADF* was not the most precise algorithm. This is due to the first upper bound found by the algorithm and its inability to prune the search tree of GREC under a very limited time constraint. Up to 500 ms, *ADF-UB* was the most precise algorithm since it started with a satisfactory *UB* that was found at the setup time. Afterwards, *ADF-UB* improved its first *UB* by outputting better ones. Under small time constraints, *FBP* was less precise than *BP*. However, when we increased the time, the gap between them shrunk and finally they got the same precision starting from 100 ms. Concerning *BS-100*, for most of the comparisons, it was unable to output feasible solutions before violating the time constraints.

On Protein-40, as illustrated in Fig. 3, as on GREC-20, *ADF* was the fastest method to output solutions, followed by *BS-1* and *BS-100*. *FBP* and *BP* solved the linear assignment problem with the help of the Hungarian and Munkres' methods, respectively. This fact prevents them from outputting solutions rapidly for relatively large graphs when time matters. Since *ADF-UB* computes *BP* as a first *UB*, its first solution is highly dependent on *BP*. When we added more time, *BP* and *FBP* outputted feasible solutions. Unlike the latter methods, *ADF*, *ADF-UB*, *BS-100* and *FBP-Beam* could still improve their solutions until the algorithm was suspended.

Fig. 4 shows the results on CMU. The same remarks as on Protein can be seen; however, the deviation of *BP* and *FBP* was high (see Fig. 4(right)). On the other hand, *ADF* and *ADF-UB* succeeded in improving the deviation as the time constraint increased. On

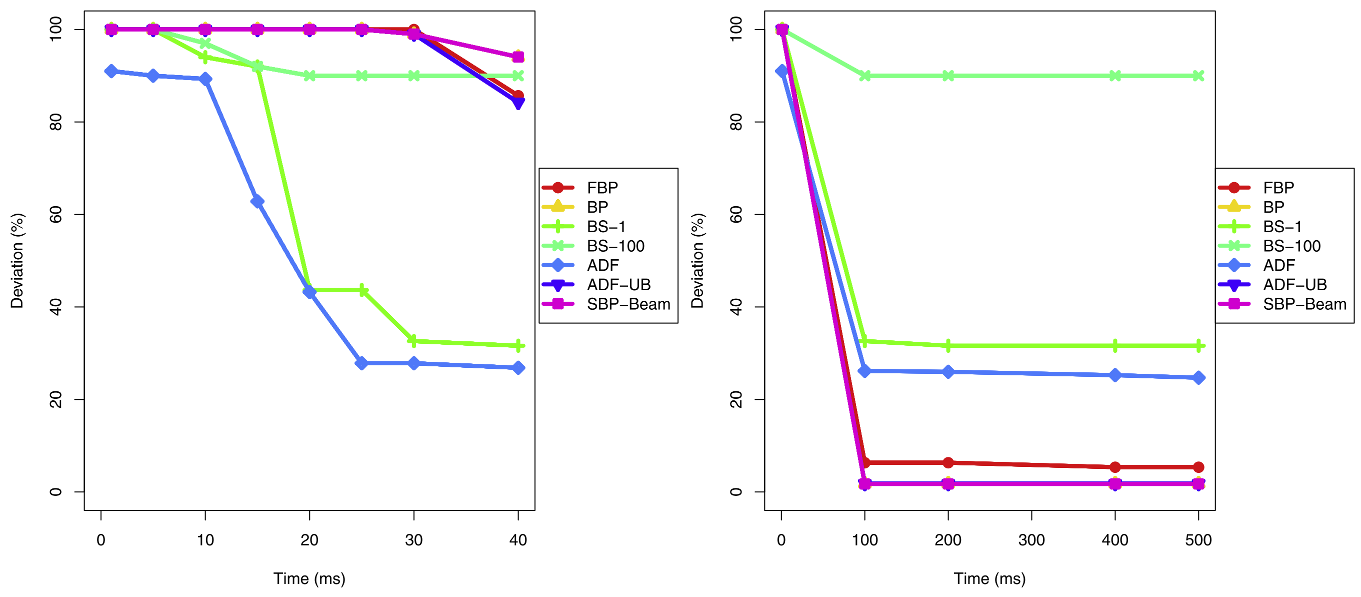

Fig. 3. Protein deviation: left (up to 40 ms), right (up to 500 ms).

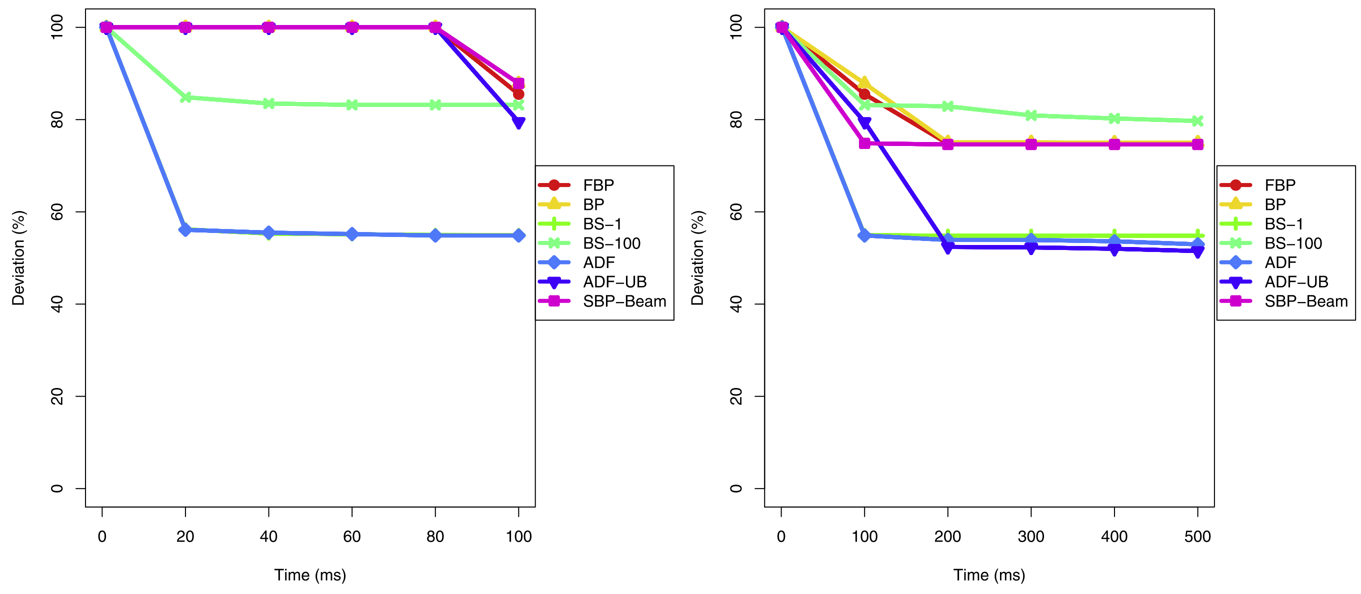

Fig. 4. CMU deviation: left (up to 40 ms), right (up to 500 ms).

Protein and CMU, *BS-100*'s deviation was quite high, as it did not find complete solutions before timing out.

As for MUTA-70, Fig. 5(a) shows that when time matters, *FBP* was surprisingly faster in outputting solutions, followed by *BP*, *SBP-Beam* and *ADF-UB*. We have argued that MUTA has low density graphs than Protein where the average $|V|/|E|$ ratio is 30.3/30.8 on MUTA and 32.6/62.1 on Protein, where $|V|$ and $|E|$ are the total numbers of vertices and edges, respectively. For this reason, solving the edges assignment problem on MUTA is faster than Protein and, thus, *FBP* and *BP* were able to output their solutions faster than *ADF*. After 40 ms, both *ADF* and *ADF-UB* beat *BP*. For instance, when C_T was equal to 400 ms, the deviation of *BP* was 45.24% whereas the deviation of *ADF* and *ADF-UB* was 35.12% and 33.02%, respectively.

To study the effect of $h(p)$, mentioned in Section 3.1.1, on *ADF* and *BS-1*, we carried out an additional experiment on MUTA-70 with plenty of time available. $h(p)$ was calculated using *BP* which is applied on the unprocessed vertices and edges analogously. Thus, several versions of *ADF* and *BS* were tested where methods with *LB* refer to the versions where the lower bound was integrated. The results in Fig. 6 demonstrates that, after 4000 ms, *BS-100-LB*, *ADF-LB* and *ADF-UB-LB* had the smallest deviation. Among these algorithms, *ADF-UB-LB* was the most accurate. One can conclude that with more time, $h(p)$ is important since it helps in converging faster to the optimal solution. *BS-100* was also unable to output feasible solutions owing to memory saturation.

Fig. 7 demonstrates the results on PAH. Since this database contains unattributed graphs, *BP*-like algorithms had a very high deviation as they failed in finding a satisfactory matching. Thus, *ADF* and *ADF-UB* got the best deviations (i.e., 31.26% and 30.44%, respectively). On this dataset, *SBP-Beam* was more precise than *BP*, where the gap between them was 8%.

For a better understanding of the performance of anytime GM algorithms, Table 4 directs the readers' attention to the average setup time, (see Section 4.1). We studied the average setup time on two databases on which anytime algorithms behaved differently.

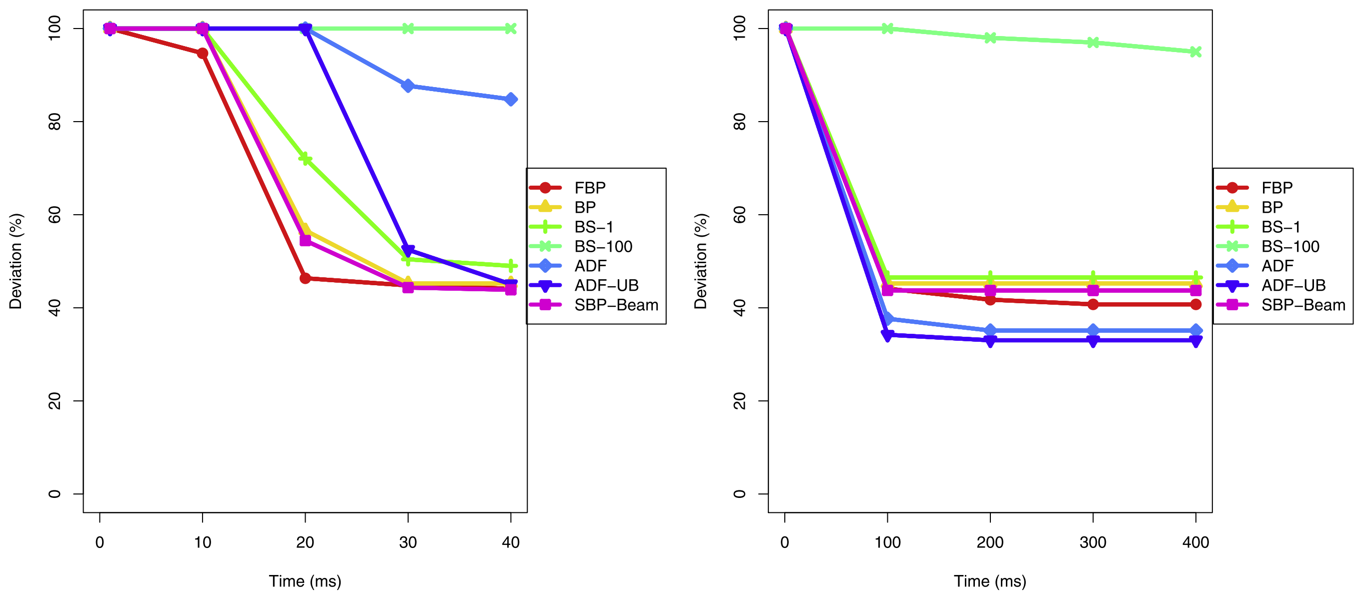

Fig. 5. MUTA deviation: left (up to 40 ms), right (up to 400 ms).

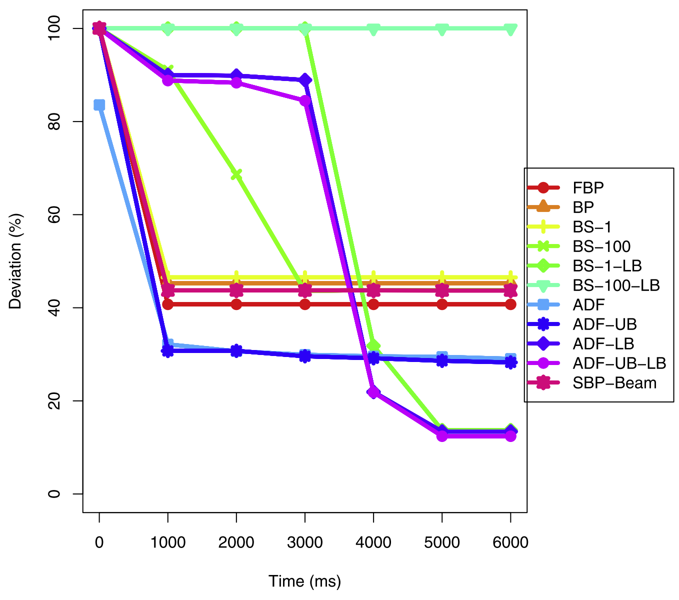

Fig. 6. MUTA Deviation: (up to 6000 ms).

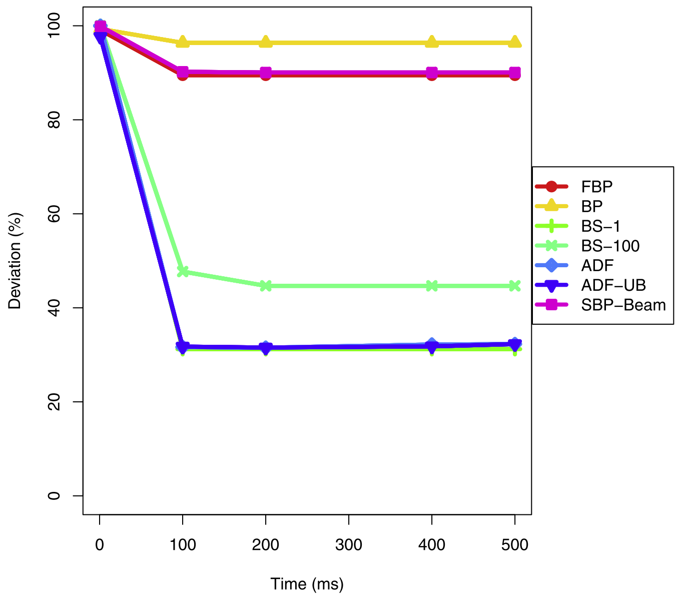

Fig. 7. PAH deviation: up to 500 ms.

Table 4
Average setup time and deviation on Protein and MUTA.

	Protein				
	SFBP	BP	BS-1	ADF	ADF-UB
Setup time (ms)	47.60	49.81	12.10	12.88	50.02
Deviation	4.344	1.789	31.597	31.490	1.789
			MUTA		
Setup time (ms)	15.60	17.55	20.02	24.70	18.35
Deviation	37.874	42.254	43.169	44.169	42.254

Table 5
The average setup time and deviation on the synthetic database.

	Density=0.1			Density=0.4		
	FBP	BP	ADF	FBP	BP	ADF
Setup Time (ms)	7169	7984	2146	383,526	391,391	5365
Deviation (%)	0	2.2	0	10.4	34.5	0

On Protein, *ADF* proved to be faster than the approximate algorithms. On average, *ADF* only needed 12.88 ms to output a solution. However, this was not the case on MUTA where *FBP* was the fastest (only 15.60 ms on average).

We have previously argued that *FBP* and *BP* are faster when graphs have low density whereas *ADF* is faster when graphs are have high density. To prove that, we carried out some experiments on the synthetic database, (see Section 5.2). Table 5 shows that *ADF* was the fastest and the most precise algorithm when increasing graph density.

7. Conclusion and perspectives

In the present paper, we have considered the problem of error-tolerant GM computation under time and memory constraints. We presented a simple approach for converting an optimal algorithm of GM into an anytime one that offers a trade-off between search time and solution quality. DFS algorithms are anytime by nature. Thus, in this paper, we proposed an anytime algorithm, referred to as *ADF*, that is based on a depth-first GM algorithm (*DF*) in [2]. *DF* does not consume so much memory. It is also able to find an initial, possibly suboptimal, solution quickly and then continues to search for improved solutions until it converges to the optimal solution. In order to convert *DF* into an anytime one, *DF* is equipped with the appropriate interruption criteria and the output is made available at anytime *t*.

The simplicity of *ADF* makes it very easy to use. It can be used not only when the optimal solution is desired, but also when time is limited. In the experiments, we focused on both the deviation when varying the timeout and the minimal time needed by anytime algorithms to get the first solution on different graph datasets. Results showed that there is a trade-off between time and quality. *FBP* and *BP* were faster when graphs were less dense whereas *ADF* was faster when graphs were denser. It is remarkable that anytime algorithms are also effective when we have some additional time, which guarantees to find better solutions. Merging *ADF* and *BP* as in *ADF-UB* is also beneficial since *ADF* can improve the solutions found by *BP*. On the selected datasets, experiments showed that *ADF* and *ADF-UB* outperformed all approximate methods by only waiting for 100 ms per graph comparison. This conclusion brings into question the usual evidences that claim that it is impossible to use optimal methods in real-world applications when matching large graphs. We conclude that *ADF* provides an attractive approach to challenging GM problems, especially when the time and memory available are limited or uncertain and when we are interested in improving the best solution found so far.

To the best of our knowledge, this work was the first attempt to introduce anytime algorithms for GM. In future work, more experiments will be conducted to understand better the effect of graph's structures on approximate and anytime algorithms. Moreover, others heuristic search methods or anytime version (like CBS [33]) can be adapted to solve the GM problem and could be compared with the method proposed in the paper. We will also propose solutions for anytime GM algorithms that can be interrupted (stopped) automatically when the quality of the actual solution is sufficient for the targeted application or when, even with much more time, the quality of the solution will not increase significantly.

References

[1] Z. Abu-Aisheh, R. Raveaux, J.-Y. Ramel, A graph database repository and performance evaluation metrics for graph edit distance, in: Graph-Based Representations in Pattern Recognition - GbRPR 2015., 2015, pp. 138–147.
[2] Z. Abu-Aisheh, R. Raveaux, J.-Y. Ramel, P. Martineau, An exact graph edit distance algorithm for solving pattern recognition problems, in: Proceedings of ICPRAM, 2015, pp. 271–278.
[3] H.A. Almohamad, S.O. Duffuaa, A linear programming approach for the weighted graph matching problem, IEEE Trans. Pattern Anal. Mach. Intell. 15 (5) (1993) 522–525.
[4] CMU., http://vasc.ri.cmu.edu/idb/html/motion. (2013).
[5] D. Conte, P. Foggia, C. Sansone, M. Vento, Thirty years of graph matching, Pattern Recog. Artif. Intell. 18 (2004) 265–298.
[6] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, Third Edition, 3rd edition, The MIT Press, 2009.
[7] P. Erdős, A. Rényi, On random graphs. I, Publ. Math. Debrecen 6 (1959) 290–297.
[8] M. Ferrer, F. Serratosa, K. Riesen, Improving bipartite graph matching by assessing the assignment confidence, Pattern Recogn. Lett. 65 (2015) 29–36.
[9] B. Gauzere, L. Brun, D. Villemin, Two new graphs kernels in chemoinformatics, Pattern Recogn. Lett. 33 (15) (2012) 2038–2047.
[10] E.A. Hansen, R. Zhou, Anytime heuristic search, J. Artif. Int. Res. 28 (1) (2007) 267–297.
[11] J.E. Hopcroft, J.K. Wong, Linear time algorithm for isomorphism of planar graphs (preliminary report), in: Proceedings of the Sixth Annual ACM Symposium on Theory of Computing, 1974, pp. 172–184.
[12] D. Justice, A. Hero, A binary linear programming formulation of the graph edit distance, IEEE Trans. Pattern Anal. Mach. Intell. 28 (2006) 1200–1214.
[13] A.L. Köll, H. Kaindl, A new approach to dynamic weighting, in: Proceedings of the 10th European Conference on Artificial Intelligence, 1992, pp. 16–17.
[14] R.E. Korf, Linear-space best-first search, Artif. Intell. 62 (1) (1993) 41–78.
[15] M. Leordeanu, M. Hebert, R. Sukthankar, An integer projected fixed point method for graph matching and map inference, in: Proceedings Neural Information Processing Systems, 2009, pp. 1114–1122.
[16] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, S. Thrun, Anytime search in dynamic graphs, Artif. Intell. 172 (2008) 1613–1643.
[17] Z. Liu, H. Qiao, GNCCP - graduated nonconvexityand concavity procedure, IEEE Trans. Pattern Anal. Mach. Intell. 36 (2014) 1258–1267.
[18] J. Munkres, Algorithms for the assignment and transportation problems, J. Soc. Ind. Appl. Math. 5 (1) (1957) 32–38.
[19] M. Neuhaus, K. Riesen, H. Bunke, Fast suboptimal algorithms for the computation of graph edit distance., SSPR. 28 (2006) 163–172.
[20] B.H. Riesen K., Approximate graph edit distance computation by means of bipartite graph matching., Image Vis. Comput. 28 (2009) 950–959.
[21] K. Riesen, Structural Pattern Recognition with Graph Edit Distance - Approximation Algorithms and Applications, Advances in Computer Vision and Pattern Recognition, Springer, 2015.
[22] K. Riesen, H. Bunke, Iam graph database repository for graph based pattern recognition and machine learning (2008) 287–297.
[23] K. Riesen, H. Bunke, Improving approximate graph edit distance by means of a greedy swap strategy 8509 (2014) 314–321.
[24] K. Riesen, S. Fankhauser, H. Bunke, Speeding up graph edit distance computation with a bipartite heuristic, in: Mining and Learning with Graphs, MLG 2007, Firence, Italy, August 1–3, 2007, Proceedings, 2007.
[25] K. Riesen, A. Fischer, H. Bunke, Improving approximate graph edit distance using genetic algorithms, 2014, pp. 63–72.
[26] R. Riesen., Classification and clustering of vector space embedded graphs., 2009 PhD thesis.
[27] F. Serratosa, Computation of graph edit distance: reasoning about optimality and speed-up, Image Vis. Comput. 40 (2015) 38–48.
[28] M. Shimbo, T. Ishida, Controlling the learning process of real-time heuristic search, Artif. Intell. 146 (1) (2003) 1–41.
[29] A. Torsello, D. Hidovic-Rowe, M. Pelillo, Polynomial-time metrics for attributed trees, IEEE Trans. Pattern Anal. Mach. Intell. 27 (2005) 1087–1099.
[30] S. Umeyama, An eigen decomposition approach to weighted graph matching problems, IEEE Trans. Pattern Anal. Mach. Intell. 10 (1988) 695–703.
[31] M. Vento, A long trip in the charming world of graphs for pattern recognition, Pattern Recogn. 48 (2) (2015) 291–301.
[32] M. Zaslavskiy, F. Bach, J. Ver, A path following algorithm for the graph matching problem, IEEE Trans. Pattern Anal. Mach. Intell. 31 (2009) 2227–2242.
[33] W. Zhang, Complete anytime beam search, in: Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence, 1998, pp. 425–430.
[34] F. Zhou, F.D. la Torre, Factorized graph matching, in: 2012 IEEE Conference on Computer Vision and Pattern Recognition, 2012, pp. 127–134.
[35] S. Zilberstein, Using anytime algorithms in intelligent systems, AI Magazine 17 (3) (1996) 73–83.
[36] S. Zilberstein, S.J. Russell, Approximate Reasoning Using Anytime Algorithms, in: S. Natarajan (Ed.), Imprecise and Approximate Computation, 1995.