Solving the Graph Edit Distance Problem with Variable Partitioning Local Search

Mostafa Darwiche[1,2](✉), Donatello Conte[1], Romain Raveaux[1], and Vincent T'kindt[2]

[1] Laboratoire d'Informatique Fondamentale et Appliquée de Tours (EA 6300), University of Tours, Tours, France
{mostafa.darwiche,donatello.conte,romain.raveaux}@univ-tours.fr

[2] Laboratoire d'Informatique Fondamentale et Appliquée de Tours (EA 6300), ERL-CNRS 6305, University of Tours, Tours, France
tkindt@univ-tours.fr

Abstract. In the world of graph matching, the *Graph Edit Distance* (GED) problem is a well-known distance measure between graphs. It has been proven to be a $\mathcal{NP}$-hard minimization problem. This paper presents an adapted version of *Variable Partitioning Local Search* (VPLS) matheuristic for solving the GED problem. The main idea in VPLS is to perform local searches in the solution space of a *Mixed Integer Linear Program* (MILP). A local search is done in a small neighborhood defined based on a set of special variables. Those special variables are selected based on a procedure that extracts useful characteristics from the instance at hand. This actually ensures that the neighborhood contains high quality solutions. Finally, the experimentation results have shown that VPLS has outperformed existing heuristics in terms of solution quality on CMU-HOUSE database.

Keywords: Graph Edit Distance · Graph Matching · Mixed Integer Linear Program · Variable Partitioning Local Search · Matheuristic

1 Introduction

Graphs are heavily involved in *Structural Pattern Recognition* (SPR). Using graphs, it is possible to model objects and patterns by considering the main components as vertices and expressing the relations between those components using edges. Moreover, graphs can store extra properties and characteristics about the pattern by assigning attributes to vertices and edges. Then, these graphs are exploited to perform object comparison and recognition [15]. In fact, this is known as *Graph Matching* (GM), which is the core of the SPR field. GM is about finding vertices and edges mappings between two graphs, from which a (dis-)similarity measure can be computed. In addition, GM covers many problems that are split into two main categories: *Exact* (EGM) and *Error-Tolerant* (ETGM). The main difference between the two categories is that EGM requires having the same topologies and attributes in graphs. While ETGM is flexible and accommodates to differences in graphs. ETGM is more preferable because it is unlikely to have the same exact graphs in real-life scenarios. Among the various problems that fall into ETGM category, the *Graph Edit Distance* (GED) problem is considered as the most popular one. Solving this problem computes a dissimilarity measure between two graphs [4]. The main idea in GED is to transform one source graph into another target graph, by applying a certain number of edit operations. Those edit operations are: substitution, insertion and deletion of a vertex/edge. Each edit operation has an associated cost. The aim when solving the GED problem is to find a set of edit operations that minimizes the total cost. This is what makes it a very complex problem to solve, which later was proven to be a $\mathcal{NP}$-hard minimization problem [19]. Despite the complex nature of the problem, it is still seen as an important one because it was shown to be a generalization to other GM problems such as the maximum common subgraph and the subgraph isomorphism [2,3]. Also, the GED has applications in many fields such as image analysis, biometrics, bio/cheminformatics, etc. [18].

Looking into the literature, numerous methods for solving the GED problem exist. They can be divided into two classes: exact and heuristics. In the first class, there are methods that solve an instance to the problem to optimality. Such methods tend to become expensive when dealing with large graphs, because of the exponential growth in complexity. The other class, however, contains heuristic methods that aim at computing a sub-optimal solution in a reasonable amount of time. For exact methods, mathematical programming is used to model the GED problem providing *Mixed Integer Linear Models* (MILP). Two formulations appear to outperform other methods: JH by Justice and Hero [11] and F3 by Darwiche et al. [8]. JH is designed to solve a sub-problem of the GED (denoted by GED^{EnA}), in which the attributes on edges are ignored. However, F3 is designed to solve instances of the general GED problem. Regarding the heuristic methods, there are plenty of them. Starting with the most famous and fastest one the Bipartite GM heuristic (denoted shortly by BP), which was developed by Riesen et al. [16]. BP breaks down the GED problem into a linear sum assignment problem that can be solved in polynomial time, using the Hungarian algorithm [14]. Later, BP has been improved in many works such as *FastBP* and *SquareBP* [17], and also it has been used in other heuristics such as *SBPBeam* [10]. Other heuristics are, for instance, *Integer Projected Fixed Point* (IPFP) and *Graduate Non Convexity and Concavity Procedure* (GNCCP) [1]. They are based on solving the *Quadratic Assignment Problem* (QAP) model for the GED problem proposed by the same authors. A recent heuristic method has been designed in [7] and referred to as *LocBra*. It is based on local searches in the solution space of a MILP formulation. This kind of heuristics on the basis of MILP formulations is known by *Matheuristics*. LocBra was shown in [6,7] to be more effective than existing heuristics (e.g. SBPBeam, IPFP and GNCCP) when dealing with instances of GED^{EnA}.

This work is an attempt to design a new matheuristic that can accomplish accurate results as LocBra but on the general problem. It is actually an adapted version of *Variable Partitioning Local Search* (VPLS) matheuristic proposed originally by Della Croce et al. [9]. The main ingredients in VPLS are: a MILP formulation which is going to be F3 and a MILP solver which is CPLEX. Then, VPLS defines neighborhoods around feasible solutions by modifying the MILP. The modified formulation will be handed over to the solver to explore the neighborhoods looking for improved solutions. The special version dedicated to the GED problem involves integrating problem-dependent information and characteristics into the neighborhood definition, which increases most likely the performance of the heuristic.

The remainder is organized as follows: Sect. 2 presents the definition of GED problem, followed with a review of $F3$ formulation. Then, Sect. 3 details the proposed heuristic. And Sect. 4 shows the results of the computational experiments. Finally, Sect. 5 highlights some concluding remarks.

2 *GED* Definition and *F*3 Formulation

2.1 *GED* Problem Definition

Given two graphs $G = (V, E, \mu, \xi)$ and $G' = (V', E', \mu', \xi')$, GED is the task of transforming the graph source G into the graph target G'. To accomplish this, GED introduces vertex and edge edit operations: $(i \rightarrow k)$ is the substitution of two vertices, $(i \rightarrow \epsilon)$ is the deletion of a vertex, and $(\epsilon \rightarrow k)$ is the insertion of a vertex, with $i \in V, k \in V'$ and ϵ refers to the empty node. The same logic goes for edges. The set of operations that reflects a valid transformation of G into G' is called a *complete edit path*, defined as $\lambda(G, G') = \{o_1, ..., o_k\}$, where o_i is an elementary vertex (or edge) edit operation and k is the number of operations. GED is then,

$$d_{min}(G, G') = \min_{\lambda \in \Gamma(G,G')} \sum_{o_i \in \lambda} \ell(o_i) \tag{1}$$

where $\Gamma(G, G')$ is the set of all complete edit paths between G and G', d_{min} represents the minimal cost obtained by a complete edit path $\lambda(G, G')$, and $\ell(.)$ is a cost function that assigns costs to elementary edit operations.

2.2 Mixed Integer Linear Program

The general MILP formulation is of the form:

$$\min_x \; c^T x \tag{2}$$

$$Ax \geq b \tag{3}$$

$$x_i \in \{0, 1\}, \forall i \in B \tag{4}$$

$$x_j \in \mathbb{N}, \forall j \in I \tag{5}$$

$$x_k \in \mathbb{R}, \forall k \in C \tag{6}$$

where $c \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$ are vectors of coefficients, $A \in \mathbb{R}^{m \times n}$ is a matrix of coefficients. x is a vector of variables to be computed. The variable index set is split into three sets (B, I, C), which stand for binary, integer and continuous, respectively. This formulation minimizes an objective function (Eq. 2) w.r.t. a set of linear inequality constraints (Eq. 3) and the bounds imposed on variables x e.g. integer or binary (Eqs. 4–6). A feasible solution is a vector x with the proper values based on their defined types, that satisfies all the constraints. The optimal solution is a feasible solution that has the minimum objective function value. This approach of modeling decision problems (i.e. problems with binary and integer variables) is very efficient, especially for hard optimization problems.

2.3 *F*3 Formulation

$F3$ is a recent MILP formulation proposed by Darwiche et al. [8], which was an improvement to an earlier version (referred to as $F2$) designed by Lerouge et al. [12]. $F3$ is a compact formulation with a set of constraints independent from the edges in the graphs. For this reason, F3 is more effective than $F2$ especially in the case of dense graphs [8]. In the following, $F3$ is detailed by defining: data, variables, objective function to minimize and constraints to satisfy.

Data. Given two graphs $G = (V, E, \mu, \xi)$ and $G' = (V', E', \mu', \xi')$, the cost functions, in order to compute the cost of each vertex/edge edit operations, are known and defined. Therefore, vertices cost matrix $[c_v]$ is computed as in Eq. 7 for every couple $(i, k) \in V \times V'$. The ϵ column is added to store the cost of deletion of i vertices, while the ϵ row stores the costs of insertion of k vertices. Following the same process, the matrix $[c_e]$ is computed for every $((i, j), (k, l)) \in E \times E'$, plus the row/column ϵ for deletion and insertion of edges.

$$c_v = \begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array} \begin{array}{cc} \begin{array}{ccccc} v_1 & v_2 & \cdots & v_{|V'|} & \epsilon \end{array} & \\ \left[\begin{array}{ccccc} c_{1,1} & c_{1,2} & \cdots & c_{1,|V'|} & c_{1,\epsilon} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,|V'|} & c_{2,\epsilon} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ c_{|V|,1} & c_{|V|,2} & \cdots & c_{|V|,|V'|} & c_{|V|,\epsilon} \\ c_{\epsilon,1} & c_{\epsilon,2} & \cdots & c_{\epsilon,|V|} & 0 \end{array}\right] & \begin{array}{c} u_1 \\ u_2 \\ \vdots \\ u_{|V|} \\ \epsilon \end{array} \end{array} \tag{7}$$

Variables. $F3$ focuses on finding the correspondences between the sets of vertices and the sets of edges. So, two sets of decision variables are needed.

- $x_{i,k} \in \{0, 1\}$ $\forall i \in V, \forall k \in V'$; $x_{i,k} = 1$ when vertices i and k are matched, and 0 otherwise.
- $y_{ij,kl} \in \{0, 1\}$ $\forall (i, j) \in E, \forall (k, l) \in E' \cup \overline{E}'$ such that $\overline{E}' = \{(l, k) : \forall (k, l) \in E'\}$; $y_{ij,kl} = 1$ when edge (i, j) is matched with (k, l), and 0 otherwise.

Objective Function. The objective function to minimize is the following.

$$\min_{x,y} \sum_{i \in V} \sum_{k \in V'} (c_v(i,k) - c_v(i,\epsilon) - c_v(\epsilon,k)) \cdot x_{i,k} \\ + \sum_{(i,j) \in E} \sum_{(k,l) \in E'} (c_e(ij,kl) - c_e(ij,\epsilon) - c_e(\epsilon,kl)) \cdot y_{ij,kl} + \gamma \tag{8}$$

The objective function minimizes the cost of assigning vertices and edges with the cost of substitution subtracting the cost of insertion and deletion. The γ, which is a constant given in Eq. 9, compensates the subtracted costs of the assigned vertices and edges. This constant does not impact the optimization algorithm and it could be removed. It is there to obtain the GED value. So at first, the function considers all vertices and edges of G as deleted and the ones of G' as inserted. Then, it solves the problem of finding the cheapest substitution assignments of vertices and edges.

$$\gamma = \sum_{i \in V} c_v(i,\epsilon) + \sum_{k \in V'} c_v(\epsilon,k) + \sum_{(i,j) \in E} c_e(ij,\epsilon) + \sum_{(k,l) \in E'} c_e(\epsilon,kl) \tag{9}$$

Constraints. $F3$ has 3 sets of constraints.

$$\sum_{k \in V'} x_{i,k} \leq 1 \ \forall i \in V \tag{10}$$

$$\sum_{i \in V} x_{i,k} \leq 1 \ \forall k \in V' \tag{11}$$

$$\sum_{(i,j) \in E} \sum_{(k,l) \in E' \cup \overline{E}'} y_{ij,kl} \leq d_{i,k} \times x_{i,k} \ \forall i \in V, \forall k \in V' \\ \text{with } d_{i,k} = \min(degree(i), degree(k)) \tag{12}$$

Constraints 10 and 11 are to make sure that a vertex can be only matched with maximum one vertex. Next, constraints 12 guarantee preserving edges matching between two couple of vertices.

3 VPLS Heuristic

3.1 Main Features of VPLS

Variable Partitioning Local Search (VPLS) is a matheuristic proposed by Della Corce et al. [9]. It aims at solving optimization problems by embedding a MILP solver into heuristic algorithms. More generally, VPLS is about performing neighborhood exploration in the solution space of a MILP formulation. To start a VPLS heuristic, two ingredients are needed: MILP formulation and MILP solver.

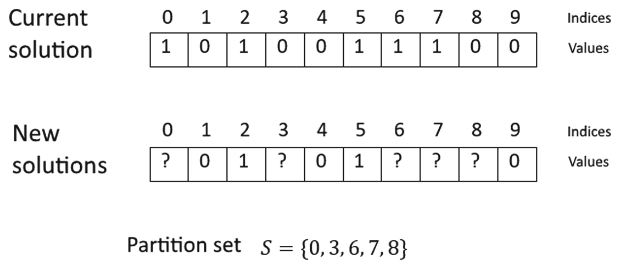

Fig. 1. Example of VPLS partitioning.

The first step in VPLS heuristic is to compute a feasible solution $\bar{X}$. Let $X_B = \{x_i | \forall i \in B\}$ be the set of binary variables and $\bar{X}_B = \{\bar{x}_i | \forall i \in B\}$ be the set of values assigned to binary variables based on $\bar{X}$. Now, assuming that there exists a partition $S \subseteq B$ of "special" binary variables. The variables in S are selected based on some defined rules, where these rules underlie some analyses and observations related to the problem. Later, a procedure is presented for selecting those special variables based on problem-dependent information and characteristics of an instance. After determining the set S, a neighborhood $N(\bar{X}, S)$ can be defined as follows:

$$N(\bar{X}, S) = \{X_B \mid x_j = \bar{x}_j, \forall j \notin S\} \tag{13}$$

The neighborhood of $\bar{X}$, then, contains all solutions of the MILP such that, they share the same values of binary variables not belonging to subset S, as in the current solution $\bar{X}_B$. Meanwhile, the variables belonging to subset S remain free. An example of variables partitioning is depicted in Fig. 1. So, the resulting restricted MILP formulation has a part of its binary variables with default values (as in the solution $\bar{X}$). At this point, the solver can be called to solve the restricted formulation looking for the optimal/best solution in the neighborhood $N(\bar{X}, S)$. The new solution is the optimal in that neighborhood, if the proof of optimality is returned by the solver. In the case where the restricted formulation is difficult, then the solver can be forced to stop and return the best solution found so far. This step stands for the search intensification in VPLS. Finally, the current solution $\bar{X}$ is updated with the new solution. To sum up, VPLS consists of three main steps:

1. Neighborhood definition around a current solution $\bar{X}$.
2. Intensifying the search in the neighborhood.
3. Updating the current solution with the new one.

The process can be repeated until a defined stopping criterion is met.

3.2 VPLS for the GED Problem

To make the heuristic suitable for the GED problem, $F3$ is selected as the main formulation. Then, A fundamental question arises when implementing VPLS is how to define the set S? Earlier, the variables in S were referred to as special variables, and this is to indicate that they should be chosen carefully. Choosing them randomly is a possibility, but there is no guarantee that the neighborhoods will contain good and diversified solutions.

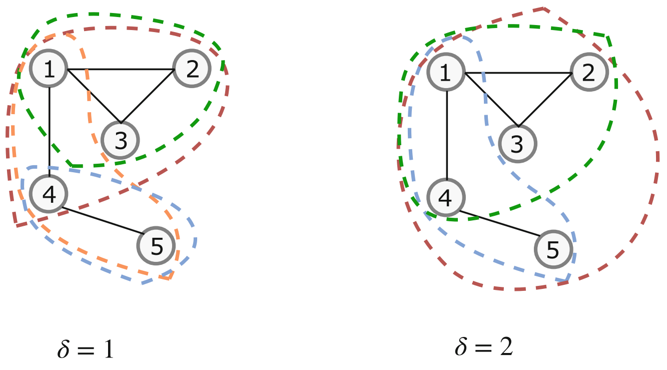

Fig. 2. Example of generating spheres for a graph. When $\delta = 1$, in red is the sphere for vertex 1, in green is the sphere for vertices 2 and 3, in orange is the sphere for vertex 4 and in blue is the sphere for vertex 5. When $\delta = 2$, in red is the sphere for vertices 1 and 4, in green is the sphere for vertices 2, 3, and in blue is the sphere for vertex 5. (Color figure online)

So, back to defining the set S, it is essential to select variables that affect the most the matching (and at the same time the objective function). Basically, only $x_{i,k}$ variables are going to be considered when defining the set S. And next, a procedure based on the notion of spheres is followed to determine S. This procedure needs two input graphs G and G' and an initial solution x^0, and it proceeds as follows:

(i) First, define the list of spheres on graph G of radius δ. For each vertex i in G, the sphere $\mathscr{S}_i$ contains all vertices j that are distant from i with at most δ edges, e.g. if $\delta = 1$, $\mathscr{S}_i$ contains all vertices connected to i with an edge. To compute how many edges are needed to go from one vertex to another, the well-known Dijkstra algorithm is used [5]. It computes the shortest path between two vertices in a graph. In fact, each sphere is a subgraph of G, containing all vertices accessible by at most δ edges, plus the edges connecting any two vertices in the sphere. Figure 2 shows an example of spheres with different δ values.

(ii) Next, compute a cost for each sphere $\mathscr{S}_i$ based on the assignments in the initial solution x^0. For example, if $\mathscr{S}_1$ for vertex u_1 contains vertices $\{u_1, u_2, u_3, u_4\}$. From the solution x^0, see to which vertex $k \in V'$ the vertex u_1 is assigned, and include the cost $c(u_1 \to v)$ to the sphere's cost. As well, check the edges that are part of sphere $\mathscr{S}_1$ and find their assignments so their costs are added to the sphere's cost. The same is done for the rest of the vertices $(\{u_2, u_3, u_4\})$.

$$c_{\mathscr{S}} = \sum_{\forall i \in \mathscr{S}} c(i \rightarrow assign(i)) + \sum_{\forall (i,j) \in (\mathscr{S} \times \mathscr{S}) \cap E} c((i,j) \rightarrow assign((i,j))), \quad (14)$$

with *assign* a function to determine vertices/edges assignments based on x^p solution. The result of this step is an array $[c_{\mathscr{S}}]$ storing costs of all spheres.

(iii) Finally, find the sphere with the highest cost in $[c_{\mathscr{S}}]$ array. Then, for every vertex i in this sphere, add all $x_{i,k}$ variables to the set S.

Steps (ii) and (iii) are called each time a new feasible solution is found to select the next sphere with the highest cost. An already selected sphere is excluded in the next iteration. This avoids selecting a sphere multiple times, and searching in the same neighborhood several times consecutively.

Once the set S is determined, the next step is to set all variables not in S to their values in the solution x^0 and the rest of the variables are left free in the MILP formulation. The solver will solve the restricted MILP formulation trying to find the best solution by setting the right values for variables in S. This is the intensification phase, that will result in a new solution x^1. Again, the spheres costs are recomputed based on x^1, and the one with the highest cost will be selected for the next iteration. An iteration, then, consists of three steps: computing and selecting the sphere to define S, defining the neighborhood based on S, intensifying the search in the neighborhood. This process is repeated until reaching some defined stopping criterion.

Finally, VPLS requires the following parameters to be set:

1. δ, is the radius of spheres.
2. *total_time_limit*, is the total running time allowed for VPLS before stopping.
3. *node_time_limit*, is the maximum running time given to the solver to solve the restricted MILP formulation.
4. *UB_time_limit*, is the running time allowed to the solver to compute an initial solution.
5. *cons_sol_max*, serves as a stopping criterion: VPLS stops when the number of consecutive intensification steps finding solutions with the same objective function values is equal to this parameter.

4 Computational Experiments

Database. Among the numerous existing databases, CMU-HOUSE database is selected in this experiment [13]. It contains 111 (attributed and undirected) graphs corresponding to $3D$-images of houses. The particularity of this database is that graphs are extracted from $3D$-images of houses, where the houses are rotated with different angles. This is interesting because it enables testing and comparing graphs representing the same house but positioned differently inside the images. The total number of instances is 660.

Experiment Settings and Comparative Heuristics. VPLS algorithm is implemented in C language. The solver CPLEX 12.7.1 is used to solve the MILP formulations. CPLEX solver is configured to use a single thread, and the rest of the parameters are set to default. Experiments are ran on a machine with Windows 7×64, Intel Xeon $E5$ 2.30 GHz, 4 cores and 8 Gb of RAM. The following heuristics are selected from the literature in the comparison: SBPBeam [10], IPFP and GNCCP [1]. Their parameters are set to the values as mentioned in the references. VPLS parameters are set empirically to the following values: $\delta = 2$, $cons\_sol\_max = 5$, $total\_time\_limit = 10\,s$, $node\_time\_limit = 2\,s$, $UB\_time\_limit = 4\,s$. For each heuristic, the following indicators are computed: t_{min}, t_{avg}, and t_{max} are the minimum, average and maximum CPU times in seconds over all instances. Correspondingly, d_{min}, d_{avg}, and d_{max} are the deviations of the solutions obtained by one heuristic, from the best solutions found by all heuristics (i.e. given an instance I and a heuristic H, deviation percentage is equal to $\frac{solution_I^H - best_I}{best_I} \times 100$, with $best_I$ the best solution for I found by all heuristics).

Table 1. VPLS vs. heuristics on CMU-HOUSE instances

	VPLS	SBPBeam	IPFP	GNCCP
t_{min}	5.76	7.54	0.03	6.85
t_{avg}	9.24	8.50	**0.18**	9.61
t_{max}	10.03	9.72	0.32	11.70
d_{min}	0.00	0.00	0.00	0.00
d_{avg}	**13.13**	330.24	313.05	336.81
d_{max}	294.41	5502.39	34308.00	32426.89
η_I	**537**	126	310	440

Results and Analysis : Based on the results reported in Table 1, VPLS seems to be the best heuristic in terms of solutions quality, with the best average deviation at 13% and η_I at 537. The difference is remarkably high (around 300%) compared to the deviations obtained by other heuristics. Even when looking at worst deviations the difference is very high. However, in terms of average running time, the fastest heuristic is IPFP with t_{avg} at 0.18 s, while other heuristics including VPLS reaches $9\,s$. Eventually, VPLS has been able to outperform the existing heuristics by obtaining very good solutions.

5 Conclusion

In this work, a VPLS heuristic is designed for solving the GED problem. This heuristic is based on performing local searches on the basis of a MILP formulation. To perform local searches, the neighborhoods are defined based on special variables determined by extracting characteristics from the instance at hand. By doing so, the performance of VPLS improves, which was shown in the experiments where VPLS outperformed the existing heuristics in terms of solution quality. This is a second matheuristic designated to solve the GED problem after the first and successful attempt with local branching [7]. Indeed, matheuristics are effective and are new ways for solving the GED problem. Meanwhile, it will be interesting to combine VPLS and local branching into one matheuristic by unifying the neighborhood definitions. As well, more evaluations and experiments need to be performed to test the methods on different kinds of graphs.

References

1. Bougleux, S., Brun, L., Carletti, V., Foggia, P., Gaüzère, B., Vento, M.: Graph edit distance as a quadratic assignment problem. Pattern Recogn. Lett. **87**, 38–46 (2017)
2. Bunke, H.: On a relation between graph edit distance and maximum common subgraph. Pattern Recogn. Lett. **18**(8), 689–694 (1997)
3. Bunke, H.: Error correcting graph matching: on the influence of the underlying cost function. IEEE Trans. Pattern Anal. Mach. Intell. **21**(9), 917–922 (1999)
4. Bunke, H., Allermann, G.: Inexact graph matching for structural pattern recognition. Pattern Recogn. Lett. **1**(4), 245–253 (1983)
5. Cormen, T.H.: Section 24.3: Dijkstra's algorithm. In: Introduction to Algorithms, pp. 595–601 (2001)
6. Darwiche, M., Conte, D., Raveaux, R., T'Kindt, V.: Graph edit distance: accuracy of local branching from an application point of view. Pattern Recogn. Lett. (2018). https://doi.org/10.1016/j.patrec.2018.03.033. http://www.sciencedirect.com/science/article/pii/S0167865518301119
7. Darwiche, M., Conte, D., Raveaux, R., T'Kindt, V.: A local branching heuristic for solving a graph edit distance problem. Comput. Oper. Res. (2018). https://doi.org/10.1016/j.cor.2018.02.002. http://www.sciencedirect.com/science/article/pii/S0305054818300339
8. Darwiche, M., Raveaux, R., Conte, D., T'Kindt, V.: Graph edit distance in the exact context. In: Bai, X., Hancock, E.R., Ho, T.K., Wilson, R.C., Biggio, B., Robles-Kelly, A. (eds.) S+SSPR 2018. LNCS, vol. 11004, pp. 304–314. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-97785-0_29
9. Croce, F.D., Grosso, A., Salassa, F.: Matheuristics: embedding MILP solvers into heuristic algorithms for combinatorial optimization problems. In: Siarry, P. (ed.) The Oxford Handbook of Innovation, Chap. 3. NOVA Publisher (2013)
10. Ferrer, M., Serratosa, F., Riesen, K.: Improving bipartite graph matching by assessing the assignment confidence. Pattern Recogn. Lett. **65**, 29–36 (2015)
11. Justice, D., Hero, A.: A binary linear programming formulation of the graph edit distance. IEEE Trans. Pattern Anal. Mach. Intell. **28**(8), 1200–1214 (2006)
12. Lerouge, J., Abu-Aisheh, Z., Raveaux, R., Héroux, P., Adam, S.: New binary linear programming formulation to compute the graph edit distance. Pattern Recogn. **72**, 254–265 (2017)
13. Moreno-García, C.F., Cortés, X., Serratosa, F.: A graph repository for learning error-tolerant graph matching. In: Robles-Kelly, A., Loog, M., Biggio, B., Escolano, F., Wilson, R. (eds.) S+SSPR 2016. LNCS, vol. 10029, pp. 519–529. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49055-7_46

14. Munkres, J.: Algorithms for the assignment and transportation problems. J. Soc. Ind. Appl. Math. **5**(1), 32–38 (1957)
15. Riesen, K.: Structural Pattern Recognition with Graph Edit Distance. Advances in Computer Vision and Pattern Recognition. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-27252-8
16. Riesen, K., Neuhaus, M., Bunke, H.: Bipartite graph matching for computing the edit distance of graphs. In: Escolano, F., Vento, M. (eds.) GbRPR 2007. LNCS, vol. 4538, pp. 1–12. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-72903-7_1
17. Serratosa, F.: Computation of graph edit distance: reasoning about optimality and speed-up. Image Vis. Comput. **40**, 38–48 (2015)
18. Stauffer, M., Tschachtli, T., Fischer, A., Riesen, K.: A survey on applications of bipartite graph edit distance. In: Foggia, P., Liu, C.-L., Vento, M. (eds.) GbRPR 2017. LNCS, vol. 10310, pp. 242–252. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58961-9_22
19. Zeng, Z., Tung, A.K., Wang, J., Feng, J., Zhou, L.: Comparing stars: on approximating graph edit distance. Proc. VLDB Endow. **2**(1), 25–36 (2009)