Accepted manuscript, published in Pattern Recognition Letters 129, pages 19-25, 2020
© 2019. This version is made available under the CC-BY-NC-SA 4.0 license: http://creativecommons.org/licenses/by-nc-sa/4.0/
Journal version available at https://doi.org/10.1016/j.patrec.2019.10.028
Code available at https://github.com/dbblumenthal/gedlib
Work supported by Région Normandie (France) under project RIN AGAC

Improved local search for graph edit distance

Nicolas Boria[a], David B. Blumenthal[b], Sébastien Bougleux[a], Luc Brun[a]

[a]Normandie Univ, ENSICAEN, UNICAEN, CNRS, GREYC, 14000 Caen, France
[b]Free University of Bozen-Bolzano, Faculty of Computer Science, Piazza Dominicani 3, 39100 Bolzano, Italy

ABSTRACT

The graph edit distance (GED) measures the dissimilarity between two graphs as the minimal cost of a sequence of elementary operations transforming one graph into another. This measure is fundamental in many areas such as structural pattern recognition or classification. However, exactly computing GED is $\mathcal{NP}$-hard. Among different classes of heuristic algorithms that were proposed to compute approximate solutions, local search based algorithms provide the tightest upper bounds for GED. In this paper, we present K-REFINE and RANDPOST. K-REFINE generalizes and improves an existing local search algorithm and performs particularly well on small graphs. RANDPOST is a general warm start framework that stochastically generates promising initial solutions to be used by any local search based GED algorithm. It is particularly efficient on large graphs. An extensive empirical evaluation demonstrates that both K-REFINE and RANDPOST perform excellently in practice.

1. Introduction

In many areas such as pattern recognition or graph classification, computing a graph (dis-)similarity measure is a central issue [1, 2, 3, 4]. One popular approach is to embed the graphs into multi-dimensional vector spaces and then to compare their vector representations [5, 6, 7]. This approach has the advantage that is allows fast computations. The main drawback is that embedding the graphs into vector spaces always comes at the price of information loss.

If local information that would be lost by the embeddings is crucial, it is recommendable to compare the graphs directly in the graph space. The graph edit distance (GED) is one of the most widely used measures for this purpose [8, 9]. GED is defined as the minimum cost of an edit path transforming one graph into another, where an edit path is a sequence of node and edge insertions, deletions, and substitutions. Equivalently, it can be defined as the minimum cost of an edit path induced by a node map that assigns nodes of the source graph to nodes of the target graph [10].

As exactly computing GED is $\mathcal{NP}$-hard [11], research has mainly focused on heuristics [12]. The development of heuristics was particularly triggered by the algorithms presented in [13] and [11], which use transformations to the linear sum assignment problem with error correction (LSAPE) [14] — a variant of the linear sum assignment problem (LSAP) where rows and columns may also be inserted and deleted — to compute upper bounds for GED. Further transformations from GED to LSAPE have been proposed in [15, 16, 17, 18, 19, 20, 21, 22].

LSAPE based heuristics are typically quite fast but yield loose upper bounds on some graphs. Tighter upper bounds for GED can be obtained by algorithms that use variants of local search. Given one or several initial node maps, local search based algorithms explore suitably defined neighborhoods to find improved node maps that induce cheaper edit paths. Several algorithms have been proposed that instantiate this paradigm: REFINE [11] varies the initial node maps via binary swaps; IPFP [23, 24, 25] computes locally optimal node maps by using a variant of the classical Frank-Wolfe algorithm [26, 27]; BP-BEAM [28] uses beam search to improve the initial node maps; and IBP-BEAM [29] further improves BP-BEAM by iteratively running it with different processing orders.

In this paper, we propose a new local search based algorithm K-REFINE and a warm start framework RANDPOST. K-REFINE generalizes and improves the existing algorithm REFINE in three respects: Firstly, K-REFINE considers not only binary swaps, but rather swaps of size up to K, where $K \in \mathbb{N}_{\geq 2}$ is a meta-parameter. Secondly, K-REFINE computes the swap costs more efficiently than REFINE, which leads to a significant gain in runtime performance. Thirdly, unlike REFINE, K-REFINE allows the improved node map to contain fewer node substitutions than the original node map, which tightens the produced upper bound.

The framework RANDPOST extends the local search paradigm and hence improves all local search based GED algorithms. In a first step, RANDPOST runs a local search algorithm from a set of initial node maps. Subsequently, RANDPOST stochastically generates a new set of initial node maps from the converged solutions, and then re-runs the local search algorithm. This process iterates until a user-specified number of iterations has been reached. Extensive experiments show that K-REFINE performs

extremely well on small to medium sized graphs, while using RANDPOST is particularly effective on larger graphs.

The remainder of this paper is organized as follows: In Section 2, we fix concepts and notations and present the general local search framework. In Section 3, we summarize existing local search algorithms. In Section 4 and Section 5, we present K-REFINE and RANDPOST. In Section 6, K-REFINE and RANDPOST are evaluated empirically. Section 7 concludes the paper. The paper extends the results published in [30], where a preliminary version of RANDPOST was presented.

2. Preliminaries

2.1. Definitions and notations

An *undirected labeled graph* G is a 4-tuple $G = (V^G, E^G, \ell_V^G, \ell_E^G)$, where V^G and E^G are sets of nodes and edges, Σ_V and Σ_E are label alphabets, and $\ell_V^G : V^G \to \Sigma_V$, $\ell_E^G : E^G \to \Sigma_E$ are labeling functions. The *dummy symbol* ϵ denotes dummy nodes and edges as well as their labels. Throughout the paper, we denote the nodes of a graph G by $V^G := \{u_i \mid i \in [|V^G|]\}$ and the nodes of a graph H by $V^H := \{v_k \mid k \in [|V^H|]\}$, where the index set $[N]$ is defined as $[N] := \{n \in \mathbb{N} \mid 1 \leq n \leq N\}$, for all $N \in \mathbb{N}$.

We denote by *assignment* a pair (u, v) in $(V^G \cup \{\epsilon\}) \times (V^H \cup \{\epsilon\})$, and we define a *node map* π as a set of assignments such that each node in V^G and V^H appears in exactly one assignment of π. The notation $(u, v) \in \pi$ is considered equivalent to both $\pi(u) = v$ and $\pi^{-1}(v) = u$, and $\Pi(G, H)$ denotes the set of all node maps between G and H. For edges $e = (u, u') \in E^G$ and $f = (v, v') \in E^H$, we introduce the short-hand notations $\pi(e) := (\pi(u), \pi(u'))$ and $\pi^{-1}(f) := (\pi^{-1}(v), \pi^{-1}(v'))$.

A node map $\pi \in \Pi(G, H)$ specifies for all nodes and edges of G and H whether they are substituted, deleted, or inserted. Each of these operations has an induced cost. An assignment (u, v) induces a node substitution with cost $c_V(u, v)$ if $u \neq \epsilon$ and $v \neq \epsilon$, a node deletion with cost $c_V(u, \epsilon)$ if $u \neq \epsilon$ and $v = \epsilon$, and a node insertion with cost $c_V(\epsilon, v)$ if $u = \epsilon$ and $v \neq \epsilon$. Similarly, a pair of assignments $((u, v), (u', v'))$ induces an edge substitution with cost $c_E((u, u'), (v, v'))$ if $(u, u') \in E^G$ and $(v, v') \in E^H$, an edge deletion with cost $c_E((u, u'), \epsilon)$ if $(u, u') \in E^G$ and $(v, v') \notin E^H$, and an edge insertion with cost $c_E(\epsilon, (v, v'))$ if $(u, u') \notin E^G$ and $(v, v') \in E^H$. The edit cost functions c_V and c_E are defined in terms of the labeling functions ℓ_V^G, ℓ_V^H, ℓ_E^G, and ℓ_E^H, i. e., nodes and edges with the same labels induce the same edit costs.

Any node map $\pi \in \Pi(G, H)$ hence induces an *edit path* P_π between G and H. The cost $c(P_\pi)$ of P_π is given as follows:

$$c(P_\pi) = \underbrace{\sum_{\substack{u \in V^G \\ \pi(u) \in V^H}} c_V(u, \pi(u))}_{\text{node substitutions}} + \underbrace{\sum_{\substack{u \in V^G \\ \pi(u) \notin V^H}} c_V(u, \epsilon)}_{\text{node deletions}} + \underbrace{\sum_{\substack{v \in V^H \\ \pi^{-1}(v) \notin V^G}} c_V(\epsilon, v)}_{\text{node insertions}}$$
$$+ \underbrace{\sum_{\substack{e \in E^G \\ \pi(e) \in E^H}} c_V(e, \pi(e))}_{\text{edge substitutions}} + \underbrace{\sum_{\substack{e \in E^G \\ \pi(e) \notin E^H}} c_E(e, \epsilon)}_{\text{edge deletions}} + \underbrace{\sum_{\substack{f \in E^H \\ \pi^{-1}(f) \notin E^G}} c_E(\epsilon, f)}_{\text{edge insertions}}$$

We can now formally define GED.

Definition 1 (GED). The *graph edit distance* (GED) between two graphs G and H is defined as $\mathrm{GED}(G, H) := \min_{\pi \in \Pi(G,H)} c(P_\pi)$.

2.2. Upper bounding GED via local search

By Definition 1, each node map $\pi \in \Pi(G, H)$ induces an upper bound $UB := c(P_\pi)$ for $\mathrm{GED}(G, H)$. Hence, a straightforward application of the local search paradigm to the problem of upper bounding GED works as follows: Given an initial node map $\pi \in \Pi(G, H)$, run a local search algorithm to obtain an improved node map π' with $c(P_{\pi'}) \leq c(P_\pi)$.

With this approach, the quality of the obtained node map π' clearly depends a lot on the initial node map π. In order to reduce this dependency, it was suggested in [31] to generate κ different initial solutions, run the local search algorithm on each of them (possibly in parallel), and return the best among the κ computed local optima. In order to reduce the computing time when parallelization is available, it was suggested in [30] to run in parallel more local searches than the number of desired local optima and to stop the whole process once the number local searches that have converged has reached the number of desired local optima. In this context, the framework runs with two parameters: κ represents the number of initial solutions, and $0 < \rho \leq 1$ is defined such that $\lceil \rho \cdot \kappa \rceil$ represents the number of desired computed local optima.

3. Existing local search algorithms for GED

3.1. The algorithm REFINE

Given an initial node map $\pi \in \Pi(G, H)$, the algorithm REFINE [11] proceeds as follows: Let $((u_s, v_s))_{s=1}^{|\pi|}$ be an arbitrary ordering of the initial node map π, and $G_\pi := (V_\pi^G \cup V_\pi^H, A_\pi)$ be an auxiliary directed bipartite graph, where $V_\pi^G := \{u_s \mid s \in [|\pi|]\}$, $V_\pi^H := \{v_s \mid s \in [|\pi|]\}$, and $A_\pi := \pi \cup \{(v_s, u_{s'}) \mid (s, s') \in [|\pi|] \times [|\pi|] \wedge s \neq s'\}$. In other words, G_π contains a forward arc for each assignment contained in π, and backward arcs between nodes in V_π^G and V_π^H that are not assigned to each other by π. A directed cycle $C \subseteq A_\pi$ in G_π with $|C| = 4$ is called swap.

For each swap $C = \{(u_s, v_s), (v_s, u_{s'}), (u_{s'}, v_{s'}), (v_{s'}, u_s)\}$, REFINE checks if the swapped node map $\pi' := (\pi \setminus \{(u_s, v_s), (u_{s'}, v_{s'})\}) \cup \{(u_s, v_{s'}), (u_{s'}, v_s)\}$ induces a smaller upper bound than π. If, at the end of the for-loop, a node map has been found that improves the upper bound, π is updated to the node map that yields the largest improvement and the process iterates. Otherwise, the current node map π is returned.

3.2. The algorithm IPFP

The algorithm IPFP [27] is a variant of the Frank-Wolfe algorithm [26] for cases where an integer solution is required. Its adaptation to GED suggested in [23, 24, 25] implicitly constructs a matrix $\mathbf{D} \in \mathbb{R}^{((|V^G|+1)\cdot(|V^H|+1)) \times ((|V^G|+1)\cdot(|V^H|+1))}$ such that $\min_{\mathbf{X} \in \Pi'(G,H)} \mathrm{vec}(\mathbf{X})^\mathsf{T} \mathbf{D}\, \mathrm{vec}(\mathbf{X}) = \mathrm{GED}(G, H)$, where $\Pi'(G, H) \subseteq \{0, 1\}^{(|V^G|+1) \times (|V^H|+1)}$ contains all node maps between G and H in matrix form and $\mathrm{vec}(\cdot)$ is a vectorization operator.

Let the cost function $c : [0, 1]^{(|V^G|+1) \times (|V^H|+1)} \to \mathbb{R}$ be defined as $c(\mathbf{X}) := \mathrm{vec}(\mathbf{X})^\mathsf{T} \mathbf{D}\, \mathrm{vec}(\mathbf{X})$. Starting from an initial node map $\mathbf{X}_0 \in \Pi'(G, H)$ with induced upper bound $UB := c(\mathbf{X}_0)$, the

algorithm initializes $\mathbf{X}' := \mathbf{X}_0$ and converges to a, possibly fractional, local minimum by repeating the five following steps:

1. Populate LSAPE instance $\mathbf{C}_k := \mathbf{D}\,\mathrm{vec}(\mathbf{X}_k)$.
2. Compute $\mathbf{B}_{k+1} \in \arg\min_{\mathbf{B}\in\Pi'(G,H)} \mathrm{vec}(\mathbf{C}_k)^\top \mathrm{vec}(\mathbf{B})$.
3. Set $\mathbf{X}' := \arg\min_{\mathbf{X}\in\{\mathbf{X}',\mathbf{B}_{k+1}\}} c(\mathbf{X})$.
4. Compute $\alpha_{k+1} := \min_{\alpha\in[0,1]} c(\mathbf{X}_k + \alpha\cdot(\mathbf{B}_{k+1} - \mathbf{X}_k))$.
5. Set $\mathbf{X}_{k+1} := \mathbf{X}_k + \alpha_{k+1}(\mathbf{B}_{k+1} - \mathbf{X}_k)$.

IPFP iterates until $|c(\mathbf{X}_k) - \mathrm{vec}(\mathbf{C}_k)^\top \mathrm{vec}(\mathbf{B}_{k+1})|/c(\mathbf{X}_k)$ is smaller than a threshold ε or a maximal number of iterations I has been reached. Subsequently, the local optimum $\mathbf{X}_{k+1}$ is projected to the closest integral solution $\widehat{\mathbf{X}}$, and the best encountered node map $\mathbf{X}' := \arg\min_{\mathbf{X}\in\{\mathbf{X}',\widehat{\mathbf{X}}\}} c(\mathbf{X})$ is returned.

3.2.1. The algorithm BP-BEAM

Given an initial node map $\pi \in \Pi(G,H)$ and a constant $B \in \mathbb{N}_{\geq 1}$, the algorithm BP-BEAM [28] starts by producing a random ordering $((u_s, v_s))_{s=1}^{|\pi|}$ of the initial node map π. BP-BEAM now constructs an improved node map π' by partially traversing an implicitly constructed tree T via beam search with beam size B. The nodes of T are tuples (π'', s), where $\pi'' \in \Pi(G,H)$ is an ordered node map and $s \in [|\pi|]$ is the depth of the tree node in T. Tree nodes (π'', s) with $s = |\pi|$ are leafs, and the children of an inner node (π'', s) are $\{(\mathtt{SWAP}(\pi'', s, s'), s+1) \mid s' \in \{s, \ldots, |\pi|\}\}$. Here, $\mathtt{SWAP}(\pi'', s, s')$ is the ordered node map obtained from π'' by swapping the assignments (u_s, v_s) and $(u_{s'}, v_{s'})$.

At initialization, BP-BEAM sets the output node map π' to the initial node map π. Furthermore, BP-BEAM maintains a priority queue q of tree nodes which is initialized as $q := \{(\pi, 1)\}$ and sorted w. r. t. non-decreasing induced edit cost of the contained node maps. As long as q is non-empty, BP-BEAM extracts the top node (π'', s) from q and updates the output node map π' to π'' if $c(P_{\pi''}) < c(P_{\pi'})$. If $s < |\pi|$, i. e., if the extracted tree node is no leaf, BP-BEAM adds all of its children to q and subsequently discards all but the first B tree nodes contained in q. Once q is empty, the cheapest encountered node map π' is returned.

3.3. The algorithm IBP-BEAM

Since the size of the priority queue q is restricted to B, which parts of the search tree T are visited by BP-BEAM crucially depends on the ordering of the initial node map π. Therefore, BP-BEAM can be improved by considering not one but several initial orderings. The algorithm IBP-BEAM suggested in [29] does exactly this. That is, given a constant number of iterations $I \in \mathbb{N}_{\geq 1}$, IBP-BEAM runs BP-BEAM with I different randomly created orderings of the initial node map π, and then returns the cheapest node map π' encountered in one of the iterations.

4. The local search algorithm K-REFINE

In this section, we extend and improve the algorithm REFINE [11] in three ways. Firstly, instead of considering only binary swaps, we make K-REFINE consider all K'-swaps for all $K' \in [K]\setminus\{1\}$, where $K \in \mathbb{N}_{\geq 2}$ is a constant (Section 4.1). Secondly, we show that for computing the induced cost $c(P_{\pi'})$ of a node map π' obtained from π via a K'-swap C, it suffices to consider the nodes and edges that are incident with C (Section 4.2). This observation yields an improved implementation of K-REFINE, which is much more efficient than the naïve implementation suggested in [11]. Thirdly, we suggest to include the dummy assignment (ϵ, ϵ) into the initial node map π before enumerating the swaps (Section 4.3). This modification allows the number of node substitutions to decrease and hence improves the quality of the obtained upper bound.

Algorithm 1: The algorithm K-REFINE.

Input: Graphs G and H, initial node map $\pi \in \Pi(G,H)$, constant $K \in \mathbb{N}_{\geq 2}$.
Output: Node map $\pi' \in \Pi(G,H)$ with $c(P_{\pi'}) \leq c(P_\pi)$.

```
1  K' := 2;                                         // initialize current size
2  C* := ∅; Δ* := 0;                                // initialize best swap
3  while Δ* < 0 ∨ K' ≤ K do                          // main loop
4      for C ∈ C_{π,K'} do                          // enumerate swaps of current size
5          Δ := SWAP-COST(π, C);                    // compute swap cost
6          if Δ < Δ* then                           // found better swap
7              C* := C; Δ* := Δ;                    // update best swap
8      if Δ* < 0 then                               // found better node map
9          π' := SWAP(π, C*, );                     // compute swapped node map
10         c(P_π') := c(P_π) + Δ*;                  // set swapped node map cost
11         π := π';                                 // update current node map
12         K' := 2;                                 // reset current swap size
13     else
14         K' := K' + 1;                            // increment current swap size
15     C* := ∅; Δ* := 0;                            // reset best swap
16 return π' := π;                                  // return improved node map
```

4.1. Generalization to swaps of size larger than two

Algorithm 1 gives an overview of the algorithm K-REFINE, which generalizes REFINE to swaps of size larger than two. Given graphs G and H, an initial node map $\pi \in \Pi(G,H)$, and a maximal swap size $K \in \mathbb{N}_{\geq 2}$, K-REFINE starts by initializing the current swap size K', the best swap $C^\star$, and the best swap cost $\Delta^\star$ as $K' := 2$, $C^\star := \emptyset$, and $\Delta^\star := 0$ (lines 1 to 2). Subsequently, K-REFINE enters its main while-loop and iterates until no improved node map has been found and the current swap size exceeds the maximal swap size (line 3).

Inside the main while-loop, the algorithm K-REFINE first enumerates the set $\mathcal{C}_{\pi,K'} := \{C \subseteq A_\pi \mid C \text{ is cycle of length } K' \text{ in } G_\pi\}$ of all K'-swaps of π (line 4). The auxiliary directed bipartite graph $G_\pi := (V_\pi^G \cup V_\pi^H, A_\pi)$ is defined as in Section 3.1 above, i. e., we have $V_\pi^G := \{u_s \mid s \in [|\pi|]\}$, $V_\pi^H := \{v_s \mid s \in [|\pi|]\}$, and $A_\pi := \pi \cup \{(v_s, u_{s'}) \in V_\pi^H \times V_\pi^G \mid (s, s') \in [|\pi|]\times[|\pi|] \wedge s \neq s'\}$, where $((u_s, v_s))_{s=1}^{|\pi|}$ is an arbitrary ordering of π.

For each K'-swap $C \in \mathcal{C}_{\pi,K'}$, let $F(C) := C \cap \pi$ and $B(C) := \{(u,v) \in (V^G \cup \{\epsilon\}) \times (V^H \cup \{\epsilon\}) \mid (v,u) \in C \setminus \pi\}$ be the sets of node assignments corresponding to forward and backwards arcs contained in C, respectively. K-REFINE computes the swap cost

$$\mathtt{SWAP\text{-}COST}(\pi, C) := c(P_\pi) - c(P_{\mathtt{SWAP}(\pi,C)}), \tag{1}$$

where $\mathtt{SWAP}(\pi, C) := (\pi \setminus F(C)) \cup \{(u,v) \in B(C) \mid (u,v) \neq (\epsilon,\epsilon)\}$ is the node map obtained from π by carrying out the swap encoded

by C (line 5). If C yields an improvement, `K-REFINE` updates the best swap $C^\star$ and the best swap cost $\Delta^\star$ (lines 6 to 7).

Once all K'-swaps $C \in \mathcal{C}_{\pi,K'}$ have been visited, `K-REFINE` checks whether one of them yields an improvement w. r. t. the current node map π (line 8). If this is the case, `K-REFINE` updates π (lines 9 to 11) and resets the current swap size to $K' := 2$ (line 12). Otherwise, K' is incremented (line 14). Subsequently, `K-REFINE` resets the best swap and the best swap cost to $C^\star := \emptyset$ and $\Delta^\star := 0$, respectively (line 15). Upon termination of the main loop, `K-REFINE` returns the current node map π (line 16).

Assume that $\texttt{SWAP-COST}(\pi, C)$ can be computed in $O(\omega)$ time (cf. Section 4.2 for details). Furthermore, let $I \in \mathbb{N}$ be the number of times `K-REFINE` finds an improved node map in line 8. Note that, if the edit costs are integral, it holds that $I \leq c(P_\pi)$, where π is `K-REFINE`'s initial node map. Proposition 1 below implies that, for all $K' \in [K] \setminus \{1\}$ and each node map $\pi \in \Pi(G, H)$, we have $|\mathcal{C}_{\pi,K'}| = O((|V^G|+|V^H|)^{K'})$. Therefore, `K-REFINE`'s overall runtime complexity is $O(I(|V^G| + |V^H|)^K\omega)$.

Proposition 1. *For each node map $\pi \in \Pi(G, H)$ and each $K' \in \mathbb{N}_{\geq 2}$, it holds that $|\mathcal{C}_{\pi,K'}| = \binom{|\pi|}{K'}(K' - 1)!$.*

Proof. The proposition immediately follows from the definition of $\mathcal{C}_{\pi,K'}$. Details are omitted due to space constraints. □

4.2. Efficient computation of swap costs

Given a node map $\pi \in \Pi(G, H)$ and a K'-swap $C \in \mathcal{C}_{\pi,K'}$, let $\pi' := \texttt{SWAP}(\pi, C)$ be the node map obtained from π by swapping the forward and backward arcs contained in C. Assume that $c(P_\pi)$ has already been computed. By line 16, the swap cost $\texttt{SWAP-COST}(\pi, C)$ can be computed naïvely by computing the induced costs $c(P_{\pi'})$ of the swapped node map and then considering the difference between $c(P_\pi)$ and $c(P_{\pi'})$. By definition of $c(P_\pi)$, this requires $O(\max\{|E^G|, |E^H|\})$ time. Since $\texttt{SWAP-COST}(\pi, C)$ has to be computed in every iteration of `K-REFINE`'s inner for-loop, it is highly desirable to implement $\texttt{SWAP-COST}(\cdot, \cdot)$ more efficiently. The following Proposition 2 provides the key ingredient of a more efficient implementation.

Proposition 2. *Let $\pi \in \Pi(G, H)$ be a node map, $K' \in \mathbb{N}_{\geq 2}$ be a constant, and $C \in \mathcal{C}_{\pi,K'}$ be a K'-swap. Furthermore, let $V^G_C := \{u \in V^G \mid \exists v \in V^H \cup \{\epsilon\} : (u, v) \in F(C)\}$, $V^H_C := \{v \in V^H \mid \exists u \in V^G \cup \{\epsilon\} : (u, v) \in F(C)\}$, $E^G_C := \{e \in E^G \mid e \cap V^G_C \neq \emptyset\}$, and $E^H_C := \{f \in E^H \mid f \cap V^H_C \neq \emptyset\}$, $\Delta := \texttt{SWAP-COST}(\pi, C)$, and $\pi' := \texttt{SWAP}(\pi, C)$. Then the following equation holds:*

$$\begin{aligned}
\Delta = &\sum_{\substack{u \in V^G_C \\ \pi'(u) \neq \epsilon}} c_V(u, \pi'(u)) + \sum_{\substack{u \in V^G_C \\ \pi'(u) = \epsilon}} c_V(u, \epsilon) + \sum_{\substack{v \in V^H_C \\ \pi'^{-1}(v) = \epsilon}} c_V(\epsilon, v) \\
&+ \sum_{\substack{e \in E^G_C \\ \pi'(e) \neq \epsilon}} c_E(e, \pi'(e)) + \sum_{\substack{e \in E^G_C \\ \pi'(e) = \epsilon}} c_E(e, \epsilon) + \sum_{\substack{f \in E^H_C \\ \pi'^{-1}(f) = \epsilon}} c_E(\epsilon, f) \\
&- \sum_{\substack{u \in V^G_C \\ \pi(u) \neq \epsilon}} c_V(u, \pi(u)) - \sum_{\substack{u \in V^G_C \\ \pi(u) \neq \epsilon}} c_V(u, \pi(u)) - \sum_{\substack{v \in V^H_C \\ \pi^{-1}(v) = \epsilon}} c_E(\epsilon, v) \\
&- \sum_{\substack{e \in E^G_C \\ \pi(e) \neq \epsilon}} c_E(e, \pi(e)) - \sum_{\substack{e \in E^G_C \\ \pi(e) = \epsilon}} c_E(e, \epsilon) - \sum_{\substack{f \in E^H_C \\ \pi^{-1}(f) = \epsilon}} c_E(\epsilon, f)
\end{aligned}$$

Proof. By construction of V^G_C and V^H_C, we have $\pi(u) = \pi'(u)$, for all $u \in V^G \setminus V^G_C$, and $\pi^{-1}(v) = \pi'^{-1}(v)$, for all $v \in V^H \setminus V^H_C$. Similarly, $\pi(e) = \pi'(e)$ and $\pi^{-1}(f) = \pi'^{-1}(f)$ holds for all $e \in E^G \setminus E^G_C$ and all $f \in E^H \setminus E^H_C$. This proves the proposition. □

Proposition 2 implies that, for computing $\texttt{SWAP-COST}(\pi, C)$, only the nodes and edges contained in V^G_C, V^H_C, E^G_C, and E^H_C must be considered. Since K' is constant, $|V^G_C|, |V^H_C| \leq K'$, $|E^G_C| \leq K' \max\deg(G)$, and $|E^H_C| \leq K' \max\deg(H)$, $\texttt{SWAP-COST}(\pi, C)$ can hence be computed in $O(\max\{\max\deg(H), \max\deg(G)\})$ time. This is a significant improvement w. r. t. naïvely computing $\texttt{SWAP-COST}(\pi, C)$ in $O(\max\{|E^G|, |E^H|\})$ time.

4.3. Improvement of upper bound via dummy assignment

For each node map $\pi \in \Pi(G, H)$, let $S(\pi) := |\{(u, v) \in \pi \mid u \neq \epsilon \wedge v \neq \epsilon\}|$ denote the number of node substitutions contained in π. Now assume that `K-REFINE` as specified in Algorithm 1 is run from an initial node map $\pi \in \Pi(G, H)$ that does not contain the dummy assignment (ϵ, ϵ). Since π and $\pi \cup \{(\epsilon, \epsilon)\}$ induce the same edit path, this assumption is likely to hold in most implementations of `K-REFINE`. The following Proposition 3 shows that, under this assumption, the search space of `K-REFINE` is restricted in the sense that it includes only node maps $\pi' \in \Pi(G, H)$ with $S(\pi') \geq S(\pi)$. This has a negative effect on the quality of the upper bound produced by `K-REFINE`, as some potentially promising node maps are excluded a priori.

Proposition 3. *Let $\pi \in \Pi(G, H)$ be a node map that satisfies $(\epsilon, \epsilon) \notin \pi$ and $\pi' \in \Pi(G, H)$ be the improved node map obtained from π by running `K-REFINE` as specified in Algorithm 1. Then it holds that $S(\pi') \geq S(\pi)$.*

Proof. Let $\pi \in \Pi(G, H)$ be a node map that satisfies $(\epsilon, \epsilon) \notin \pi$, $K' \in \mathbb{N}_{\geq 2}$ be a constant, and $C \in \mathcal{C}_{\pi,K'}$ be a K'-swap. By definition of $B(C)$, we have $(\epsilon, \epsilon) \notin \texttt{SWAP}(\pi, C)$. Therefore, the proposition follows by induction on the number of times `K-REFINE` finds an improved node map in line 8, if we can show that $S(\texttt{SWAP}(\pi, C)) \geq S(\pi)$. To show this inequality, we define $S^\epsilon_F(C) := |\{(u, v) \in F(C) \mid u = \epsilon \wedge v = \epsilon\}|$ and $S^\epsilon_B(C) := |\{(u, v) \in B(C) \mid u = \epsilon \wedge v = \epsilon\}|$. It is easy to see that we have $S(\texttt{SWAP}(\pi, C)) = S(\pi) + S^\epsilon_B(C) - S^\epsilon_F(C)$. Since $(\epsilon, \epsilon) \notin \pi$, we additionally know that $S^\epsilon_F(C) = 0$. We hence obtain $S(\texttt{SWAP}(\pi, C)) = S(\pi) + S^\epsilon_B(C) \geq S(\pi)$, as required. □

The proof of Proposition 3 tells us how we have to modify `K-REFINE` in order to ensure that node maps with fewer node substitutions than the initial node map are contained its search space: We simply have to update the current node map π as $\pi := \pi \cup \{(\epsilon, \epsilon)\}$ before enumerating all K'-swaps $C \in \mathcal{C}_{\pi,K'}$ in line 4 of Algorithm 1. This modification is particularly important if the edit costs are non-metric, i. e., if it can happen that deleting plus inserting is cheaper than substituting.

5. The framework `RANDPOST`

In this section, we present `RANDPOST`, a framework that can be used to improve any local search based algorithm for upper bounding GED. Intuitively, `RANDPOST` iteratively runs a given

Algorithm 2: The framework RANDPOST.

Input: Graphs G and H, constants $\kappa \in \mathbb{N}_{\geq 1}$, $L \in \mathbb{N}$, $\rho \in (0, 1]$, and $\eta \in [0, 1]$, local search algorithm ALG, initial node map set $\mathcal{S}_0 \subseteq \Pi(G, H)$ with $|\mathcal{S}_0| = \kappa$, lower bound LB for $\mathrm{GED}(G, H)$.
Output: Upper bound UB for $\mathrm{GED}(G, H)$.

```
1 S'_0 := ALG(S_0, ρ);              // run local search on initial node maps
2 UB := min_{π'∈S'_0} c(P_π');      // set first upper bound
3 M := 0_{(|V^G|+1)×(|V^H|+1)};     // initialize scores matrix
4 for r ∈ [L] do                    // main loop
5     M := UPD-SCORES(M, S'_{r-1}, η, LB, UB); // update scores
6     S_r := GEN-NODE-MAPS(M, κ);   // generate node maps
7     S'_r := ALG(S_r, ρ); // run local search on new node maps
8     UB := min{UB, min_{π'∈S'_r} c(P_π')}; // update upper bound
9 return UB ;                       // return upper bound
```

local search algorithm. In each iteration, previously computed locally optimal node maps are combined stochastically to obtain new promising initial node maps to be used in the next iteration.

Algorithm 2 provides an overview of the framework. Given a set of initial node maps $\mathcal{S}_0 \subseteq \Pi(G, H)$ with $|\mathcal{S}_0| = \kappa$, a constant $\rho \in (0, 1]$, and a local search algorithm ALG, RANDPOST computes a set $\mathcal{S}'_0 \subseteq \Pi(G, H)$ of improved node maps with $|\mathcal{S}'_0| = \lceil \rho \cdot \kappa \rceil$ by (parallelly) running ALG on all initial node maps and terminating once $\lceil \rho \cdot \kappa \rceil$ runs have converged (line 1). Subsequently, the upper bound UB is initialized as the cost of the cheapest induced edit path encountered so far (line 2). Note that, up to this point, RANDPOST is equivalent to the local search framework with multi-start described in Section 2.2 above.

RANDPOST now initializes a matrix $\mathbf{M} \in \mathbb{R}^{(|V^G|+1)\times(|V^H|+1)} := \mathbf{0}_{(|V^G|+1)\times(|V^H|+1)}$ that contains scores $m_{i,k}$ for each possible node assignment $(u_i, v_k) \in (V^G \cup \{\epsilon\}) \times (V^H \cup \{\epsilon\})$ (line 3). The score for each substitution $(u_i, v_k) \in V^G \times V^H$ is represented by the value $m_{i,k}$, while the scores for the deletion (u_i, ϵ) and the insertion (ϵ, v_k) are represented by the values $m_{i,|V^H|+1}$ and $m_{|V^G|+1,k}$, respectively. Throughout the algorithm, $\mathbf{M}$ is maintained in such a way that $m_{i,k}$ is large just in case the corresponding node assignment appears in many cheap locally optimal node maps.

After initializing $\mathbf{M}$, RANDPOST carries out L iterations of its main for-loop, where $L \in \mathbb{N}$ is a meta-parameter (lines 4 to 8). Inside the rth iteration, RANDPOST starts by updating the scores matrix $\mathbf{M}$ by calling UPD-SCORES$(\mathbf{M}, \mathcal{S}'_{r-1}, \eta, LB, UB)$, where $\mathbf{M}$ is the current scores matrix, $\mathcal{S}'_{r-1} \subseteq \Pi(G, H)$ is the set of improved node maps obtained from the previous iteration, $\eta \in [0, 1]$ is a meta-parameter used to give greater weight to cheap node maps, LB is a previously computed lower bound for $\mathrm{GED}(G, H)$, and UB is the current upper bound (line 5). Let the matrix $\mathbf{M}' \in \mathbb{R}^{(|V^G|+1)\times(|V^H|+1)}$ be defined as $\mathbf{M}' :=$ UPD-SCORES$(\mathbf{M}, \mathcal{S}'_{r-1}, \eta, LB, UB)$. Then $\mathbf{M}'$ is given as

$$\mathbf{M}' := \mathbf{M} + \sum_{\pi' \in \mathcal{S}'_{r-1}} \left[(1 - \eta) + \eta \frac{UB - LB}{c(P_{\pi'}) - LB} \right] \mathbf{X}',$$

where $\mathbf{X}' \in \{0, 1\}^{(|V^G|+1)\times(|V^H|+1)}$ is the matrix representation of the improved node map $\pi' \in \mathcal{S}'_{r-1}$, i. e., for all $u_i \in V^G$ and all $v_k \in V^H$, we have $x'_{i,k} = 1$ just in case $(u_i, v_k) \in \pi'$, $x'_{i,|V^H|+1} = 1$ just in case $(u_i, \epsilon) \in \pi'$, and $x'_{|V^G|+1,k} = 1$ just in case $(\epsilon, v_k) \in \pi'$. If $\eta = 0$, $m_{i,k}$ represents the number of converged local optima that contain the corresponding assignment. If $\eta > 0$, assignments that appear in node maps with lower costs receive higher scores.

Once $\mathbf{M}$ has been updated, RANDPOST creates a new κ-sized set $\mathcal{S}_r \subseteq \Pi(G, H)$ of initial node maps by calling GEN-NODE-MAPS$(\mathbf{M}, \kappa)$ (line 6). GEN-NODE-MAPS$(\mathbf{M}, \kappa)$ works as follows: For each of the first $|V^G|$ rows $\mathbf{M}_i$ of $\mathbf{M}$, RANDPOST draws a column $k \in [|V^H| + 1]$ from the distribution encoded my $\mathbf{M}_i$. If $k = |V^H| + 1$, the node deletion (u_i, ϵ) is added to the node map π that is being constructed. Otherwise, the substitution (u_i, v_k) is added to π, the score $m_{j,k}$ is temporarily set to 0 for all $j \in [|V^G|] \setminus [i]$, and the column k is marked as covered. Once all nodes of G have been processed, node insertions (ϵ, v_k) are added to π for all uncovered columns $k \in [|V^H|]$. This process is repeated until κ different node maps have been created.

After creating the set $\mathcal{S}_r$ of new initial node maps, RANDPOST constructs a new $\lceil \rho \cdot \kappa \rceil$-sized set $\mathcal{S}'_r \subseteq \Pi(G, H)$ of improved node maps by (parallelly) running the local search algorithm ALG on the initial node maps contained in $\mathcal{S}_r$ and terminating once $\lceil \rho \cdot \kappa \rceil$ runs have converged (line 7). Subsequently, the upper bound is updated as the minimum of the current upper bound and the cost of the cheapest edit path induced by one of the newly computed improved node maps (line 8). Finally, RANDPOST returns the best encountered upper bound (line 9).

Table 1. Properties of test datasets.

	GREC	FP	PROTEIN	MUTA
max./avg. $\lvert V^G \rvert$	11.5/26	5.4/26	126/32.6	30.3/417
max./avg. $\lvert E^G \rvert$	12.2/30	4.4/25	149/62.1	30.8/112

6. Empirical evaluation

In order to empirically evaluate K-REFINE and RANDPOST, extensive tests were conducted on four standard datasets from the IAM Database Repository [32, 33]: MUTA, PROTEIN, GREC, and FP (cf. Table 1). For all datasets, we tested on the metric edit costs suggested in [33]. For MUTA, we additionally defined non-metric edit costs by setting the costs of node and edge deletions and insertions to 1, and setting the costs of node and edge substitutions to 3 (the resulting dataset is denoted as MUTA-N). For each dataset, subsets of 50 graphs were chosen randomly, and upper bounds for GED were computed for each pair of graphs in the subsets, as well as for each graph and a shuffled copy of itself. In the following, d, $\hat{d}$, and t denote the average upper bound, the average upper bound between graphs and their shuffled copies, and the average runtime in seconds, respectively. Note that the test metric $\hat{d}$ gives us a hint to how close to optimality each algorithm is, as the optimal value, namely 0, is known. All methods were implemented using the GEDLIB library and were run in 20 parallel threads.[1]

[1] Sources and datasets: https://github.com/dbblumenthal/gedlib/.

Table 2. `K-REFINE` **vs.** `REFINE` **without** `RANDPOST`**.**

dataset	`REFINE`		`2-REFINE`		`3-REFINE`	
	d	t	d	t	d	t
GREC	859.92	$2.46 \cdot 10^{-2}$	857.89	$\mathbf{1.15 \cdot 10^{-2}}$	**857.12**	$3.85 \cdot 10^{-2}$
FP	**2.82**	$5.34 \cdot 10^{-4}$	**2.82**	$\mathbf{4.49 \cdot 10^{-4}}$	**2.82**	$1.58 \cdot 10^{-3}$
PROTEIN	295.61	$3.43 \cdot 10^{-1}$	295.55	$\mathbf{9.07 \cdot 10^{-2}}$	**295.29**	$4.89 \cdot 10^{-1}$
MUTA	74.12	$1.22 \cdot 10^{-1}$	74.12	$\mathbf{3.92 \cdot 10^{-2}}$	**73.61**	$1.87 \cdot 10^{-1}$
MUTA-N	49.49	$1.14 \cdot 10^{-1}$	49.11	$\mathbf{3.58 \cdot 10^{-2}}$	**48.44**	$1.81 \cdot 10^{-1}$

We tested two versions `2-REFINE` and `3-REFINE` of our local search algorithm `K-REFINE`, which use swaps of maximum size two and three, respectively. We compared them to the existing local search algorithms `REFINE`, `IPFP`, `BP-BEAM`, and `IBP-BEAM`. As suggested in [28] and [29], we set the beam size employed by `BP-BEAM` and `IBP-BEAM` to 5 and the number of iterations employed by `IBP-BEAM` to 20. `IPFP` was run with convergence threshold set to 10^{-3} and maximum number of iterations set to 100, as proposed in [25]. In order to evaluate `RANDPOST`, we ran each algorithm with $\kappa := 40$ initial solutions. Indeed, experiments reported in [31] show that on all tested datasets, using more initial solutions does not bring a significant decrease of the estimated GED. We also varied the pair of meta-parameters (L, ρ) on the set $\{(0, 1), (1, 0.5), (3, 0.25), (7, 0.125)\}$. Recall that L is the number of `RANDPOST` loops and ρ is defined such that each iteration produces exactly $\lceil \rho \cdot \kappa \rceil$ locally optimal node maps. Therefore, our setup ensures that each configuration produces exactly 40 local optima. For each algorithm and each dataset, we conducted pre-tests where we varied the penalty parameter η on the set $\{n/10 \mid n \in \mathbb{N}_{\leq 10}\}$, and then picked the value of η for the main experiments that yielded the best average upper bound across all `RANDPOST` configurations.

6.1. `K-REFINE` vs. `REFINE`

In a first series of experiments, we compared the versions `2-REFINE` and `3-REFINE` of our improved and generalized local search algorithm `K-REFINE` to the baseline algorithm `REFINE`. All algorithms were run without `RANDPOST`, i. e., with $(L, \rho) = (0, 1)$, and the tests were carried out on a computer using an Intel Xeon E5-2620 v4 2.10GHz CPU. Table 2 shows the results. By comparing t(`2-REFINE`) and t(`REFINE`), we see that efficiently computing the swap costs as suggested in Section 4.2 indeed significantly improves the runtime performance. Unsurprisingly, the speed-up is especially large on the datasets PROTEIN and MUTA containing the larger graphs. Comparing d(`2-REFINE`) and d(`REFINE`) shows that the inclusion of the dummy assignment proposed in Section 4.3 slightly improves the quality of the produced upper bound. As expected, the percentual improvement is largest on the dataset MUTA-N with non-metric edit costs. Finally, we observe that running `K-REFINE` with swaps of size three slightly improves the upper bounds on all datasets, but significantly increases the runtime of the algorithm.

6.2. Behaviour of `RANDPOST` framework

In a second series of experiments, we evaluated the behaviour of `RANDPOST` by running each algorithm with four different pairs of meta-parameters $(L, \rho) \in \{(0, 1), (1, 0.5), (3, 0.25), (7, 0.125)\}$. We remind that the case $(L, \rho) = (0, 1)$ amounts to a basic multi-start framework with no `RANDPOST` loop. An additional subset of 10 graphs having exactly 70 nodes was extracted from MUTA and is denoted by MUTA-L. The tests were run on a computer using an Intel(R) Xeon E5-2640 v4 2.4GHz CPU. Figure 1 visualizes the results. Since `2-REFINE` was always faster and more accurate than the baseline `REFINE` (cf. Section 6.1), we do not show plots for `REFINE`. Table 3 provides detailed numerical data for the MUTA-L subset, which turned out to be the subset with the highest variability in distances and computing times.

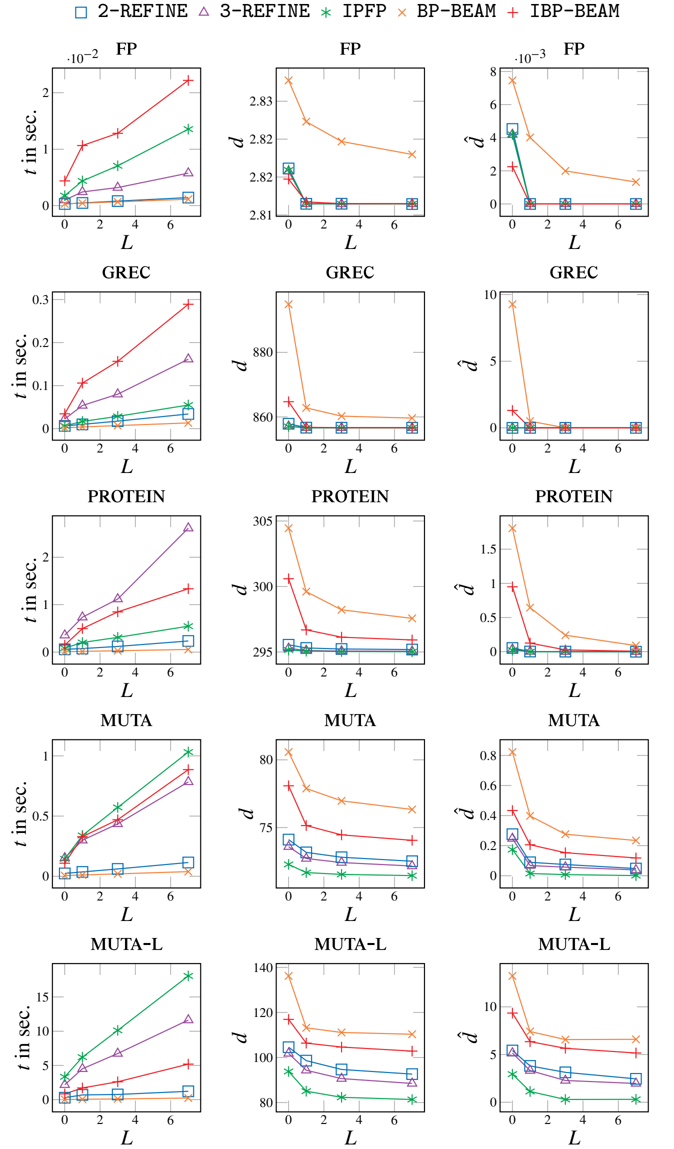

Fig. 1. Effect of `RANDPOST` **on local search algorithms.**

Figure 1 indicates that, on the datasets FP, GREC, and PROTEIN containing small graphs, near-optimility is reached by most algorithms when run with $L \geq 1$ number of `RANDPOST` loops. In these contexts, our algorithm `2-REFINE` with `RANDPOST` configuration $(L, \rho) = (1, 0.5)$ provides the best tradeoff between runtime and accuracy, as it reaches the same accuracy as best algorithms, and, in terms of runtime, outperforms all algorithms

Table 3. Detailed experimental results on MUTA-L.

(L,ρ)	2-REFINE			3-REFINE			IPFP			BP-BEAM			IBP-BEAM		
	d	$\hat{d}$	t	d	$\hat{d}$	t	d	$\hat{d}$	t	d	$\hat{d}$	t	d	$\hat{d}$	t
$(0,1)$	104.57	5.42	$2.85 \cdot 10^{-1}$	101.82	5.16	2.18	93.81	2.94	3.34	136.21	13.25	$2.95 \cdot 10^{-2}$	116.92	9.35	$8.46 \cdot 10^{-1}$
$(1,0.5)$	98.61	3.81	$6.96 \cdot 10^{-1}$	94.34	3.32	4.51	85.05	1.13	6.21	113.18	7.41	$5.81 \cdot 10^{-2}$	106.39	6.35	1.71
$(3,0.25)$	94.69	3.13	$7.61 \cdot 10^{-1}$	90.64	2.28	6.74	82.34	0.29	$1.01 \cdot 10^{1}$	111.12	6.57	$1.11 \cdot 10^{-1}$	104.64	5.65	2.61
$(7,0.125)$	92.63	2.45	1.22	88.49	1.97	$1.16 \cdot 10^{1}$	81.36	0.29	$1.81 \cdot 10^{1}$	110.32	6.59	$2.35 \cdot 10^{-1}$	102.84	5.16	5.19

except for BP-BEAM by approximately one order of magnitude. The only faster algorithm BP-BEAM computes much more expensive node maps, even in the RANDPOST settings with higher number of loops. We also note that our algorithms 2-REFINE and 3-REFINE are already among the best local search algorithms when run in a simple multi-start setting without RANDPOST (i. e., when $L = 0$), both in terms of distance and computing time.

On the datasets MUTA and MUTA-L containing larger graphs, the behavior of the RANDPOST framework appears clearly and independently of the local search algorithm it is applied to. In all cases, a higher number of RANDPOST loops — and lower number of computed solutions per loop — leads to a higher computation time (the computation time is approximately doubled whenever the number of loops is doubled), and to a lower average distance. In other words, the framework RANDPOST provides a very useful algorithmic tool in situations where some time can be dedicated to compute tight upper bounds on big graphs.

7. Conclusions

In this paper, we proposed K-REFINE, an improved and generalized version of the local search based GED algorithm REFINE, and suggested the general framework RANDPOST, which stochastically generates promising initial solutions to tighten the upper bounds produced by all local search algorithms. Both K-REFINE and RANDPOST perform excellently in practice: On small graphs, K-REFINE is among the algorithms computing the tightest upper bounds and, in terms of runtime, clearly outperforms all existing algorithms that yield similar accuracy. On larger graphs, K-REFINE provides a very good tradeoff between runtime and accuracy, as it is only slightly less accurate but much faster than the most accurate algorithms. The framework RANDPOST is particularly effective on larger graphs, where it significantly improves the upper bounds of all local search algorithms.

References

[1] D. Conte, P. Foggia, C. Sansone, M. Vento, Thirty years of graph matching in pattern recognition, Int. J. Pattern Recognit. Artif. Intell. 18 (3) (2004) 265–298. doi:10.1142/S0218001404003228.

[2] P. Foggia, G. Percannella, M. Vento, Graph matching and learning in pattern recognition in the last 10 years, Int. J. Pattern Recognit. Artif. Intell. 28 (1) (2014) 1450001:1–1450001:40. doi:10.1142/S0218001414500013.

[3] M. Vento, A long trip in the charming world of graphs for pattern recognition, Pattern Recognit. 48 (2) (2015) 291–301. doi:10.1016/j.patcog.2014.01.002.

[4] L. Brun, P. Foggia, M. Vento, Trends in graph-based representations for pattern recognition, Pattern Recognit. Lett.In press. doi:10.1016/j.patrec.2018.03.016.

[5] X. Bai, E. R. Hancock, R. C. Wilson, A generative model for graph matching and embedding, Computer Vis. Image Underst. 113 (7) (2009) 777–789. doi:10.1016/j.cviu.2009.01.004.

[6] Graph characteristics from the heat kernel trace, Pattern Recognit. 42 (11) (2009) 2589–2606. doi:10.1016/j.patcog.2008.12.029.

[7] X. Bai, E. R. Hancock, R. C. Wilson, Geometric characterization and clustering of graphs using heat kernel embeddings, Image Vis. Comput. 28 (6) (2010) 1003–1021. doi:10.1016/j.imavis.2009.05.011.

[8] H. Bunke, G. Allermann, Inexact graph matching for structural pattern recognition, Pattern Recognit. Lett. 1 (4) (1983) 245–253. doi:10.1016/0167-8655(83)90033-8.

[9] A. Sanfeliu, K.-S. Fu, A distance measure between attributed relational graphs for pattern recognition, IEEE Trans. Syst. Man Cybern. 13 (3) (1983) 353–362. doi:10.1109/TSMC.1983.6313167.

[10] D. B. Blumenthal, J. Gamper, On the exact computation of the graph edit distance, Pattern Recognit. Lett., in press. doi:10.1016/j.patrec.2018.05.002.

[11] Z. Zeng, A. K. H. Tung, J. Wang, J. Feng, L. Zhou, Comparing stars: On approximating graph edit distance, PVLDB 2 (1) (2009) 25–36. doi:10.14778/1687627.1687631.

[12] D. B. Blumenthal, N. Boria, J. Gamper, S. Bougleux, L. Brun, Comparing heuristics for graph edit distance computation, VLDB J., in press. doi:10.1007/s00778-019-00544-1.

[13] K. Riesen, H. Bunke, Approximate graph edit distance computation by means of bipartite graph matching, Image Vis. Comput. 27 (7) (2009) 950–959. doi:10.1016/j.imavis.2008.04.004.

[14] S. Bougleux, B. Gaüzère, D. B. Blumenthal, L. Brun, Fast linear sum assignment with error-correction and no cost constraints, Pattern Recognit. Lett., in press. doi:10.1016/j.patrec.2018.03.032.

[15] W. Zheng, L. Zou, X. Lian, D. Wang, D. Zhao, Efficient graph similarity search over large graph databases, IEEE Trans. Knowl. Data Eng. 27 (4) (2015) 964–978. doi:10.1109/TKDE.2014.2349924.

[16] D. B. Blumenthal, J. Gamper, Improved lower bounds for graph edit distance, IEEE Trans. Knowl. Data Eng. 30 (3) (2018) 503–516. doi:10.1109/TKDE.2017.2772243.

[17] D. B. Blumenthal, S. Bougleux, J. Gamper, L. Brun, Ring based approximation of graph edit distance, in: S+SSPR, 2018, pp. 293–303. doi:10.1007/978-3-319-97785-0_28.

[18] V. Carletti, B. Gaüzère, L. Brun, M. Vento, Approximate graph edit distance computation combining bipartite matching and exact neighborhood substructure distance, in: GbRPR, 2015, pp. 188–197. doi:10.1007/978-3-319-18224-7_19.

[19] B. Gaüzère, S. Bougleux, K. Riesen, L. Brun, Approximate graph edit distance guided by bipartite matching of bags of walks, in: S+SSPR, 2014, pp. 73–82. doi:10.1007/978-3-662-44415-3_8.

[20] D. B. Blumenthal, J. Gamper, Correcting and speeding-up bounds for non-uniform graph edit distance, in: ICDE, 2017, pp. 131–134. doi:10.1109/ICDE.2017.57.

[21] K. Riesen, H. Bunke, A. Fischer, Improving graph edit distance approximation by centrality measures, in: ICPR, 2014, pp. 3910–3914. doi:10.1109/ICPR.2014.671.

[22] X. Cortés, F. Serratosa, C. F. Moreno-García, On the influence of node centralities on graph edit distance for graph classification, in: GbRPR, 2015, pp. 231–241. doi:10.1007/978-3-319-18224-7_23.

[23] S. Bougleux, B. Gaüzère, L. Brun, Graph edit distance as a quadratic program, in: ICPR, 2016, pp. 1701–1706. doi:10.1109/ICPR.2016.7899881.

[24] S. Bougleux, L. Brun, V. Carletti, P. Foggia, B. Gaüzère, M. Vento, Graph edit distance as a quadratic assignment problem, Pattern Recognit. Lett. 87 (2017) 38–46. doi:10.1016/j.patrec.2016.10.001.

[25] D. B. Blumenthal, E. Daller, S. Bougleux, L. Brun, J. Gamper, Quasimetric graph edit distance as a compact quadratic assignment problem, in: ICPR, 2018, pp. 934–939. doi:10.1109/ICPR.2018.8546055.

[26] M. Frank, P. Wolfe, An algorithm for quadratic programming, Nav. Res. Logist. Q. 3 (1-2) (1956) 95–110. doi:10.1002/nav.3800030109.

[27] M. Leordeanu, M. Hebert, R. Sukthankar, An integer projected fixed point method for graph matching and MAP inference, in: NIPS, 2009, pp. 1114–1122.

[28] K. Riesen, A. Fischer, H. Bunke, Combining bipartite graph matching and beam search for graph edit distance approximation, in: ANNPR 2014, 2014, pp. 117–128. doi:10.1007/978-3-319-11656-3_11.

[29] M. Ferrer, F. Serratosa, K. Riesen, A first step towards exact graph edit distance using bipartite graph matching, in: GbRPR, 2015, pp. 77–86. doi:10.1007/978-3-319-18224-7_8.

[30] N. Boria, S. Bougleux, L. Brun, Approximating GED using a stochastic generator and multistart IPFP, in: S+SSPR, 2018, pp. 460–469. doi:10.1007/978-3-319-97785-0_44.

[31] É. Daller, S. Bougleux, B. Gaüzère, L. Brun, Approximate graph edit distance by several local searches in parallel, in: ICPRAM, 2018, pp. 149–158. doi:10.5220/0006599901490158.

[32] K. Riesen, H. Bunke, IAM graph database repository for graph based pattern recognition and machine learning, in: S+SSPR, 2008, pp. 287–297. doi:10.1007/978-3-540-89689-0_33.

[33] K. Riesen, H. Bunke, Graph Classification and Clustering Based on Vector Space Embedding, World Scientific, 2010. doi:10.1142/7731.