Clique-Based Encodings for Graph Edit Distance

Paulius Dilkas

27th August 2017

Abstract

Graph edit distance (GED) is a common way to quantify the similarity between two graphs. We propose two ways to encode it into a clique-based problem with additional constraints (with weighted vertices and edges and with weighted vertices only). We model and compare the two encodings with a model based on the original definition of the problem using constraint programming. We propose a new branch-and-bound algorithm with a novel heuristic based on one of the encodings, evaluate its performance on public datasets, and compare our results with experimental results for state-of-the-art exact GED algorithms. We conclude by listing several improvements that are yet to be explored.

1 Introduction

Comparing how similar two objects are is an important task in Pattern Recognition. Although it is common to represent objects as vectors, encoding them as graphs can be advantageous. For instance, graphs allow for a more natural encoding of relationship information between different subparts (relative position, distance, etc.). The problem of conceptualising similarity between two graphs is called graph matching (GM) and it has two main branches: exact and error-tolerant. While exact methods solve problems such as (sub)graph isomorphism and maximum common subgraph, error-tolerant GM allows to directly match graphs that are not isomorphic to each other and provides a number, identifying how good the matching is. One such approach is called graph edit distance (GED) and is defined in Section 2.

2 Problem definition and proposed solutions

In this section we introduce all the ideas and definitions used in this paper. Section 2.1 introduces the definition of graph edit distance. Section 2.2 defines a constraint programming (CP) model corresponding to the main definition of the problem, which acts both as a way to check our understanding of the problem and as a baseline to compare other models to. Section 2.3 introduces two clique-based GED encodings together with CP models for them. Finally, Section 2.4 explains the proposed algorithm.

2.1 The definition of graph edit distance

Most of the definitions in this section have been adapted from [1].

Definition 1. An *attributed graph* $G = (V, E, L_V, L_E, \mu, \zeta)$ is a 6-tuple where:

- V is a set of vertices,
- $E \subseteq V \times V$ is a set of edges,
- L_V is a set of vertex attributes,
- L_E is a set of edge attributes,

- $\mu : V \to L_V$ is a vertex labelling function,
- $\zeta : E \to L_E$ is an edge labelling function.

Throughout this paper, ϵ will denoted an empty (non-existant) vertex or edge.

Definition 2. Given a graph $G = (V, E)$, we define its edge function $e : (V \cup \{\epsilon\})^2 \to E \cup \{\epsilon\}$ such that for all $v, w \in V$,

$$e(\epsilon, w) = e(v, \epsilon) = e(\epsilon, \epsilon) = \epsilon$$

and $e(v, w) = \epsilon$ if there is no edge between v and w. Otherwise $e(v, w) \in E$ is that edge.

Definition 3. A function $f : V_1 \cup \{\epsilon\} \to V_2 \cup \{\epsilon\}$ is an *error-tolerant graph matching* from $G_1 = (V_1, E_1, L_{V_1}, L_{E_1}, \mu_1, \zeta_1)$ to $G_2 = (V_2, E_2, L_{V_2}, L_{E_2}, \mu_2, \zeta_2)$ if

1. considering only non-empty vertices, $f : V_1 \to V_2$ is bijective,
2. $\forall u_1, u_2 \in V_1,\ f(u_1) \neq \epsilon \implies f(u_1) \neq f(u_2)$,
3. $\forall v_1, v_2 \in V_2,\ f^{-1}(v_1) \neq \epsilon \implies f^{-1}(v_1) \neq f^{-1}(v_2)$.

Definition 4. A *graph edit operation* for an attributed graph $G = (V, E, L_V, L_E, \mu, \zeta)$ is one of the following:

- vertex insertion: $V' = V \cup \{u_i\}$ for $u_i \notin V$ (denoted $\epsilon \to u_i$),
- vertex deletion: $V' = V \setminus \{u_i\}$ for $u_i \in V$ (denoted $u_i \to \epsilon$),
- vertex substitution: redefine μ so that $\mu(u_i) = l_i \in L_V$ (denoted $u_i \to v_i$, where $\mu(v_i) = l_i$),
- edge insertion: $E' = E \cup \{e(u_i, u_j)\}$ for $e(u_i, u_j) \notin E$, $u_i, u_j \in V$ (denoted $\epsilon \to e_k$ or $\epsilon \to e(u_i, u_j)$),
- edge deletion: $E' = E \setminus \{e(u_i, u_j)\}$ for $e(u_i, u_j) \in E$ (denoted $e_k \to \epsilon$ or $e(u_i, u_j) \to \epsilon$),
- edge substitution: redefine ζ so that $\zeta(e(u_i, u_j)) = l_l \in L_E$ (denoted $e_k \to e_l$ or $e(u_i, u_j) \to e(v_i, v_j)$, where $\zeta(e_l) = \zeta(e(v_i, v_j)) = l_l$).

The set of all graph edit operations will be denoted by $\mathcal{E}$.

Definition 5. Given two graphs $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$, $c : \mathcal{E} \to \mathbb{R}$ is a valid *cost function* if the following conditions are satisfied:

- all costs are non-negative: $\forall ed_i \in \mathcal{E},\ c(ed_i) \geq 0$,
- insertion and deletion costs are positive (otherwise the same vertex/edge can be added and deleted any number of times),
- triangle inequalities: for all edit operations $u, v, w \in \mathcal{E}$,
  - $c(u, w) \leq c(u, v) + c(v, w)$,
  - $c(u, \epsilon) \leq c(u, v) + c(v, \epsilon)$,
  - $c(\epsilon, w) \leq c(\epsilon, v) + c(v, w)$.

For notational convenience we also assume that $c(\epsilon, \epsilon) = 0$.

Figure 1: One possible way to edit graph (a) into (b) is by inserting vertex 3 and matching everything else

Definition 6. Let $f : V_1 \to V_2$ be an error-tolerant GM between graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ and let $c : \mathcal{E} \to \mathbb{R}$ be an arbitrary cost function. The *matching cost function* of f is defined as

$$c(f) = \overbrace{\sum_{\substack{u_i \in V_1 \\ f(u_i) \in V_2}} c(u_i, f(u_i))}^{\text{vertex substitutions}} + \overbrace{\sum_{\substack{u_i \in V_1 \\ f(u_i) = \epsilon}} c(u_i, \epsilon)}^{\text{vertex deletions}} + \overbrace{\sum_{\substack{v_k \in V_2 \\ f^{-1}(v_k) = \epsilon}} c(\epsilon, v_k)}^{\text{vertex insertions}}$$
$$+ \overbrace{\sum_{\substack{e(u_i,u_j) \in E_1 \\ e(f(u_i),f(u_j)) \in E_2}} c(e(u_i, u_j), e(f(u_i), f(u_j)))}^{\text{edge substitutions}} + \overbrace{\sum_{\substack{e(u_i,u_j) \in E_1 \\ e(f(u_i),f(u_j)) = \epsilon}} c(e(u_i, u_j), \epsilon)}^{\text{edge deletions}}$$
$$+ \overbrace{\sum_{\substack{e(v_k,v_l) \in E_2 \\ e(f^{-1}(v_k),f^{-1}(v_l)) = \epsilon}} c(\epsilon, e(v_k, v_l))}^{\text{edge insertions}}.$$

Definition 7. A set $\{ed_1, \ldots, ed_k\}$ of k edit operations ed_i that transform a graph G_1 completely into another graph G_2 is called a *(complete) edit path* $\lambda(G_1, G_2)$ between G_1 and G_2. A partial edit path refers to a subset of $\{ed_1, \ldots, ed_k\}$. The set of all complete edit paths is denoted $\gamma(G_1, G_2)$.

For example, consider the graphs in Figure 1. Then

$$\{u_1 \to v_1, u_2 \to v_2, \epsilon \to v_3, e(u_1, u_2) \to e(v_1, v_2)\} \tag{$\star$}$$

is a complete edit path from (a) to (b). Note that instead of matching vertices this way, we could have matched them in 5 other ways or chosen to delete vertices/edges from (a) and insert the same number of vertices/edges to (b). Also note that it might be tempting to assume that a substitution is always cheaper than the sum of a deletion and an insertion, but that need not be the case. We will use this example (with the same edit path) in Section 2.3 to illustrate how the same problem can be modeled in different ways.

Definition 8. A (complete or partial) edit path λ for graphs $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$ is called *minimal* if all of the following conditions are satisfied:

$$\forall u \in V_1, \, |\{(u \to v) \in \lambda : v \in V_2 \cup \{\epsilon\}\}| \leq 1, \tag{M1}$$

$$\forall v \in V_2, \, |\{(u \to v) \in \lambda : u \in V_1 \cup \{\epsilon\}\}| \leq 1, \tag{M2}$$

$$\forall e \in E_1, \, |\{(e \to f) \in \lambda : f \in E_2 \cup \{\epsilon\}\}| \leq 1, \tag{M3}$$

$$\forall f \in E_2, \, |\{(e \to f) \in \lambda : e \in E_1 \cup \{\epsilon\}\}| \leq 1, \tag{M4}$$

$$\lambda \subseteq (\{u \to v : u \in V_1 \cup \{\epsilon\}, v \in V_2 \cup \{\epsilon\}\} \cup \{e \to f : e \in E_1 \cup \{\epsilon\}, f \in E_2 \cup \{\epsilon\}\}) \setminus \{\epsilon \to \epsilon\}. \tag{M5}$$

Definition 9. Let $G_1 = (V_1, E_1, L_{V_1}, L_{E_1}, \mu_1, \zeta_1)$ and $G_2 = (V_2, E_2, L_{V_2}, L_{E_2}, \mu_2, \zeta_2)$ be two attributed graphs and let $c : \mathcal{E} \to \mathbb{R}$ be the cost function. The *graph edit distance* between G_1 and G_2 is defined as

$$d_{\lambda_{\min}}(G_1, G_2) = \min_{\lambda \in \gamma(G_1, G_2)} \sum_{ed_i \in \lambda} c(ed_i).$$

Lemma 1. *Let $G_1 = (V_1, E_1, L_{V_1}, L_{E_1}, \mu_1, \zeta_1)$ and $G_2 = (V_2, E_2, L_{V_2}, L_{E_2}, \mu_2, \zeta_2)$ be two attributed graphs and let $c : \mathcal{E} \to \mathbb{R}$ be the cost function. There exists a minimal $\lambda \in \gamma(G_1, G_2)$ such that*

$$\sum_{ed_i \in \lambda} c(ed_i) = d_{\lambda_{min}}(G_1, G_2), \tag{†}$$

i.e., the graph edit distance between G_1 and G_2 can always be achieved with a minimal edit path.

Proof. Consider a complete edit path $\lambda' \in \gamma(G_1, G_2)$ satisfying condition (†) that is not minimal. Then it must break one of the conditions in Definition 8. Suppose it is (M1). Then $\exists u \in V_1$ s.t. $|\{(u \to v) \in \lambda : v \in V_2 \cup \{\epsilon\}\}| > 1$. Since $u \in V_1$ is unique, this can only happen if $(u_0 \to u) \in \lambda'$ for some u_0 (possibly ϵ). Replace $\{u_0 \to u, u \to v\} \subseteq \lambda'$ by $u_0 \to v$. According to Definition 5, this will keep the total cost the same (since the cost of λ' is already minimised). Repeat until the condition is satisfied. M2-M4 can be handled analogously.

Suppose λ' does not satisfy (M5). Consider each operation $ed_i \in \lambda'$ outside the union of sets in (M5). Suppose it is an insertion. Then it must be of the form $\epsilon \to v_1$ for $v_1 \notin V_2 \cup E_2$. Since this operation does not leave G_1 transformed into G_2, after a number of substitutions $v_1 \to v_2, v_2 \to v_3, \ldots, v_{n-1} \to v_n$, v_n is either deleted or substituted into a member of $V_2 \cup E_2$. Deletion is impossible because by the definition of a cost function, removing this sequence of operations would reduce the cost of λ'. Hence the sequence must end with a substitution $v_n \to v'$. But then, using the triangle inequalities in Definition 5, we can replace $\{\epsilon \to v_1, v_1 \to v_2, \ldots, v_{n-1} \to v_n, v_n \to v'\}$ with just $\epsilon \to v'$ without changing the cost. The cases for deletions and substitutions are left as exercises for the reader. After the procedure above is executed on each operation in λ' which is not in $\{u \to v : u \in V_1 \cup \{\epsilon\}, v \in V_2 \cup \{\epsilon\}\} \cup \{e \to f : e \in E_1 \cup \{\epsilon\}, f \in E_2 \cup \{\epsilon\}\}$, the resulting edit path satisfies all conditions in Definition 8 and thus is minimal. □

Corollary 1. *Let $G_1 = (V_1, E_1, L_{V_1}, L_{E_1}, \mu_1, \zeta_1)$ and $G_2 = (V_2, E_2, L_{V_2}, L_{E_2}, \mu_2, \zeta_2)$ be two attributed graphs and let $c : \mathcal{E} \to \mathbb{R}$ be the cost function. Then there exists a complete minimal edit path $\lambda \in \gamma(G_1, G_2)$ such that*

$$\sum_{ed_i \in \lambda} c(ed_i) = d_{\lambda_{min}}(G_1, G_2),$$

$$|\lambda| \leq |V_1| + |V_2| + |E_1| + |E_2|.$$

Proof. The cases for operations on vertices and edges can be analysed separately. Let λ_v, λ_e be a partition of λ such that all operations on vertices are in λ_v and all operations on edges are in λ_e. Since λ satisfying the first property must exist by Lemma 1,

$$\lambda_v \subseteq \{u \to v : u \in V_1 \cup \{\epsilon\}, v \in V_2 \cup \{\epsilon\}\}.$$

Then λ_v can be partitioned into

$$\{(u \to v) \in \lambda_v : v \in V_2 \cup \{\epsilon\}\}, \quad \forall u \in V_1 \cup \{\epsilon\}.$$

For all $u \in V_1$,

$$|\{(u \to v) \in \lambda_v : v \in V_2 \cup \{\epsilon\}\}| \leq 1$$

by (M1). Hence

$$\left| \bigcup_{u \in V_1} \{(u \to v) \in \lambda_v : v \in V_2 \cup \{\epsilon\}\} \right| \leq |V_1|. \tag{1}$$

The only remaining set from the partition of λ_v is

$$\{(\epsilon \to v) \in \lambda_v : v \in V_2\},$$

but clearly

$$|\{(\epsilon \to v) \in \lambda_v : v \in V_2\}| \leq |V_2|. \tag{2}$$

Therefore, combining (1) and (2),

$$|\lambda_v| \leq |V_1| + |V_2|.$$

The case for λ_e can be analyzed in exactly the same way, giving

$$|\lambda| \leq |V_1| + |V_2| + |E_1| + |E_2|.$$

□

Definition 10. Let $G = (V, E, L_V \subseteq \mathbb{R}, L_E \subseteq \mathbb{R}, \mu, \zeta)$ be a graph with weights on vertices and edges. Then we define the weight function for induced subgraphs $w : 2^V \to \mathbb{R}$ as

$$w(C) = \sum_{v \in C} \mu(v) + \sum_{v,w \in C} \zeta(e(v, w)).$$

If $L_E = \emptyset$ (i.e., only vertices are weighted), then

$$w(C) = \sum_{v \in C} \mu(v).$$

Definition 11. Let $G = (V, E, L_V \subseteq \mathbb{R}, L_E \subseteq \mathbb{R}, \mu, \zeta)$ be a graph with weights on vertices and edges. A *minimum weight clique* is

$$\arg\min_{C \subseteq V} w(C).$$

We will usually be interested in minimum weight cliques satisfying additional conditions.

2.2 A constraint programming model

To check if our definition of the problem is the same as in the literature and to compare our answers with a baseline solution, a simple CP model was implemented in MiniZinc.

```
include "globals.mzn";

int: n1;
int: n2;
array[1..n1,1..n1] of 0..1: adjacent1;
array[1..n2,1..n2] of 0..1: adjacent2;
array[1..n2] of float: vertexInsertionCost;
array[1..n1,0..n2] of float: vertexSubstitutionCost;
array[1..n1,1..n1] of float: edgeDeletionCost;
array[1..n2,1..n2] of float: edgeInsertionCost;
array[1..n1,1..n1,1..n2,1..n2] of float: edgeSubstitutionCost;

array[1..n1] of var 0..n2: map; % 0 means deleted
array[1..n2] of var 0..n1: inverseMap; % 0 means inserted
var float: distance;

constraint alldifferent_except_0(map);
```

```
constraint alldifferent_except_0(inverseMap);
constraint forall(i in 1..n1 where map[i] != 0)(inverseMap[map[i]] == i);
constraint forall(j in 1..n2 where inverseMap[j] != 0)(
    map[inverseMap[j]] == j);
constraint distance == sum(i in 1..n1)(vertexSubstitutionCost[i,map[i]])
    + sum(j in 1..n2)(if inverseMap[j] == 0 then vertexInsertionCost[j]
        else 0 endif)
    + sum(i in 1..n1, j in 1..i-1 where adjacent1[i,j] == 1)(
        if map[i] == 0 \/ map[j] == 0 \/ adjacent2[map[i],map[j]] == 0
        then edgeDeletionCost[i,j]
        else edgeSubstitutionCost[i,j,map[i],map[j]] endif)
    + sum(i in 1..n2, j in 1..i-1 where adjacent2[i,j] == 1)(
        if inverseMap[i] == 0 \/ inverseMap[j] == 0 \/
        adjacent1[inverseMap[i],inverseMap[j]] == 0
        then edgeInsertionCost[i,j] else 0 endif);

solve minimize distance;
```

For a graph edit distance problem from $G_1 = (V_1, E_1, L_{V_1}, L_{E_1}, \mu_1, \zeta_1)$ to $G_2 = (V_2, E_2, L_{V_2}, L_{E_2}, \mu_2, \zeta_2)$, `n1` $= |V_1|$, `n2` $= |V_2|$, `adjacent1` and `adjacent2` are the adjacency matrices of graphs G_1 and G_2 respectively, and the rest of the data encodes parts of the cost function $c : \mathcal{E} \to \mathbb{R}$. Vertex deletion cost is in the 0th column of `vertexSubstitutionCost`. Unfortunately, due to inaccuracies in working with floating-point numbers and/or bugs in gecode, this model fails on some problem instances with non-integer costs.

2.3 Clique Encodings

We introduce two encodings. Section 2.3.1 defines a graph where operation cost information is encoded as weights on both vertices and edges, while in Section 2.3.2 only vertices are weighted. Similar ideas with compatibility graphs have been widely explored (and successful) for several types of maximum common subgraph problems [7].

2.3.1 Putting weights on both vertices and edges

Let $G_1 = (V_1, E_1, L_{V_1}, L_{E_1}, \mu_1, \zeta_1)$ and $G_2 = (V_2, E_2, L_{V_2}, L_{E_2}, \mu_2, \zeta_2)$ be two arbitrary attributed graphs and let $c : \mathcal{E} \to \mathbb{R}$ be an arbitrary cost function. We will construct a new graph such that finding a minimum weight clique on it would be equivalent to finding the graph edit distance between G_1 and G_2. Construct a new weighted graph $G_3 = (V_3, E_3, L_{V_3} \subseteq \mathbb{R}, L_{E_3} \subseteq \mathbb{R}, \mu_3, \zeta_3)$ such that:

- V_3 has a 1-to-1 correspondence with the set of edit operations on vertices: $\{\epsilon \to v_i : v_i \in V_2\} \cup \{v_i \to v_j : v_i \in V_1, v_j \in V_2\} \cup \{v_i \to \epsilon : v_i \in V_1\}$,
- there is an edge between two vertices if and only if they do not share a common vertex in V_1 or V_2,
- for an arbitrary vertex edit operation $v_i \to v_j$ (with $v_i \in V_1 \cup \{\epsilon\}, v_j \in V_2 \cup \{\epsilon\}$) corresponding to $v \in V_3$, $\mu_3(v) = c(v_i, v_j)$,
- for an edge ϕ between vertices corresponding to edit operations $v_i \to v_j$, $v_k \to v_l$ (where $v_i, v_k \in V_1 \cup \{\epsilon\}$, $v_j, v_l \in V_2 \cup \{\epsilon\}$),

$$\zeta_3(\phi) = \begin{cases} \min(c(e(v_i, v_k), e(v_j, v_l)), c(e(v_i, v_k), \epsilon) + c(\epsilon, e(v_j, v_l))), & \text{if } e(v_i, v_k) \in E_1, e(v_j, v_l) \in E_2, \\ c(e(v_i, v_k), e(v_j, v_l)) & \text{otherwise.} \end{cases}$$

All operations on edges of G_1 and G_2 are encoded into edges in G_3, hence we only need to choose operations on vertices (represented by vertices of G_3) and operations on edges can be inferred from edges of G_3 that are part of the clique. Figure 2 shows the resulting G_3 for graphs from Figure 1.

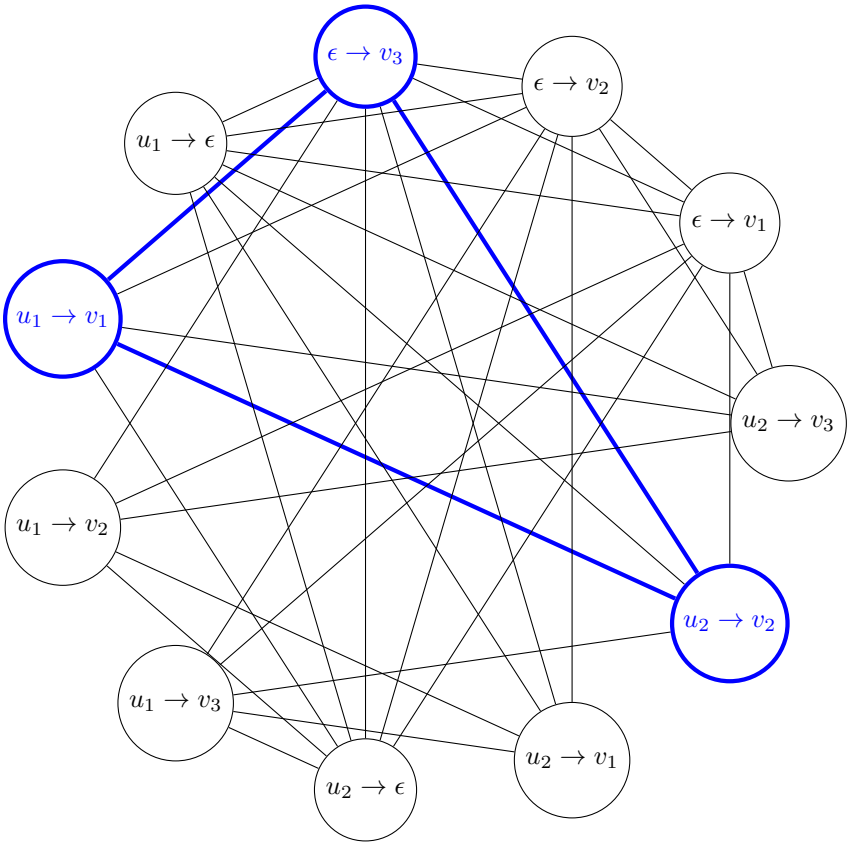

Figure 2: Vertex-edge-weights encoding of the example from Figure 1 with the clique corresponding to the edit path ($\star$) in blue

Furthermore, we must make sure that exactly one edit operation is performed on each vertex. Having multiple edit operations per vertex is already impossible because vertices of G_3, representing operations that have vertices of G_1 or G_2 in common, are not adjacent in G_3 by definition. Enforcing the need to pick an operation for each vertex of G_1 and G_2 requires new constraints, easily encoded in this CP model (which in the following sections will be referred to as the vertex-edge-weights model):

```
include "globals.mzn";

int: v1;
int: v2;
int: n = (v1+1)*(v2+1)-1; % number of vertices
array[1..n] of float: vertexWeights;
array[1..n,1..n] of float: edgeWeights;
array[1..n,1..n] of 0..1: adjacent; % adjacency matrix
array[1..n] of var 0..1: clique; % whether a vertex is part of the clique
var float: distance;

% clique property
constraint forall(i in 1..n, j in 1..i-1)(
    adjacent[i,j] == 0 -> clique[i] + clique[j] <= 1);
% we must pick a vertex from each independent set
constraint forall(i in 1..v1)(sum(j in 0..v2)(clique[i*(v2+1)+j]) >= 1);
constraint forall(j in 1..v2)(sum(i in 0..v1)(clique[i*(v2+1)+j]) >= 1);
constraint distance == sum(i in 1..n)(clique[i] * vertexWeights[i])
    + sum(i in 1..n, j in 1..i-1)(clique[i] * clique[j] * edgeWeights[i,j]);

solve minimize distance;
```

Theorem 1. *Suppose $C \subseteq V_3$ is a minimum weight clique in G_3 such that*

$$\forall v \in V_1, \exists u \in C \text{ s.t. } u \text{ represents operation } v \to v' \text{ for some } v' \in V_2 \cup \{\epsilon\}, \tag{C1}$$

$$\forall v \in V_2, \exists u \in C \text{ s.t. } u \text{ represents operation } v' \to v \text{ for some } v' \in V_1 \cup \{\epsilon\}. \tag{C2}$$

Then the set of edit operations λ corresponding to vertices in C and all edges between them is a complete minimal edit path with minimum total cost.

Proof. First we are going to prove that λ is minimal. Conditions (M1) and (M2) are satisfied because vertices in G_3 corresponding to operations $u \to v_1$, $u \to v_2$ for $u \in V_1$, $v_1, v_2 \in V_2$, $v_1 \neq v_2$ are not adjacent and thus cannot be in the same clique. To prove condition (M3), consider an edge $e(u_i, u_j) \in E_1$. Operations with it can only be represented by edges between vertices in V_3 representing operations $u_i \to v_i$, $u_j \to v_j$ for some $v_i, v_j \in V_2 \cup \{\epsilon\}$. For there to be more than one operation with $e(u_i, u_j)$, there would need to be two operations in λ of the form

$$u_i \to v, u_i \to v' \text{ for some } v, v' \in V_2 \cup \{\epsilon\} \text{ or}$$

$$u_j \to v, u_j \to v' \text{ for some } v, v' \in V_2 \cup \{\epsilon\},$$

which is impossible by definition of G_3. Condition (M4) can be handled similarly. For Condition (M5), the set of edit operations stemming from vertices of G_3 is already equal to

$$\{u \to v : u \in V_1 \cup \{\epsilon\}, v \in V_2 \cup \{\epsilon\}\} \setminus \{\epsilon \to \epsilon\}$$

and the definition of ζ_3 mentions no operations other than insertions, deletions, and substitutions from edges in E_1 to edges in E_2, thus (M5) is satisfied and λ is minimal.

Suppose λ is not a complete edit path. Then there is a vertex or edge in G_1 or G_2 which is not part of any operation in λ. Having such a vertex would contradict condition (C1) or condition (C2). Therefore it must be an edge. Pick an arbitrary edge $e(u_i, u_j) \in E_1$. We want to prove that it must be part of some operation in λ. Pick $u_c, v_c \in C$ representing operations $u_i \to v_i, u_j \to v_j$ for some $v_i, v_j \in V_2 \cup \{\epsilon\}$, respectively. They must exist by (C1). Consider the edge $e(u_c, v_c) \in E_3$. If $e(v_i, v_j) \notin E_2$ (or is not a valid edge), then

$$\zeta(e(u_c, v_c)) = c(e(u_i, u_j), e(v_i, v_j)) = c(e(u_i, u_j), \epsilon),$$

which means that $e(u_c, v_c)$ represents a deletion. Otherwise

$$\zeta(e(u_c, v_c)) = \min(c(e(u_i, u_j), e(v_i, v_j)), c(e(u_i, u_j), \epsilon) + c(\epsilon, e(v_i, v_j))),$$

which makes it either a substitution or a deletion and an insertion. Either way $e(u_c, v_c) \in E_3$ represents an operation on $e(u_i, u_j)$. Case for $e(u_i, u_j) \in E_2$ can be handled similarly. Therefore λ must be complete.

Let λ' be any complete minimal edit path and let λ'_v, λ'_e be a partition of λ', separating operations on vertices from operations on edges. Since λ' is minimal, operations in λ'_v can be mapped to vertices in a clique $C' \subseteq V_3$. Edge operations can be mapped to edges between them according to the second part of this proof. Then

$$w(C') = \sum_{v \in C'} \mu_3(v) + \sum_{v,w \in C'} \zeta_3(e(v, w)) = \sum_{ed_i \in \lambda'_v} c(ed_i) + \sum_{ed_i \in \lambda'_e} c(ed_i) = \sum_{ed_i \in \lambda'} c(ed_i),$$

hence minimising w will always produce a complete minimal edit path with minimum total cost. □

2.3.2 Weights on vertices only

Let $G_1 = (V_1, E_1, L_{V_1}, L_{E_1}, \mu_1, \zeta_1)$ and $G_2 = (V_2, E_2, L_{V_2}, L_{E_2}, \mu_2, \zeta_2)$ be two arbitrarily attributed graphs. Create a new graph $G_3 = (V_3, E_3, L_{V_3} \subseteq \mathbb{R}, \emptyset, \mu_3)$ with weights on vertices and no edge labels defined as follows:

- V_3 has a 1-to-1 correspondence with the set of all possible edit operations $\{\epsilon \to v_i : v_i \in V_2\} \cup \{v_i \to v_j : v_i \in V_1, v_j \in V_2\} \cup \{v_i \to \epsilon : v_i \in V_1\} \cup \{\epsilon \to e_i : e_i \in E_2\} \cup \{e_i \to e_j : e_i \in E_1, e_j \in E_2\} \cup \{e_i \to \epsilon : e_i \in E_1\}$.
- E_3 can be defined in terms of what edges it does not contain:
  - Similarly to the previous encoding, for all $v_i, v_k \in V_1 \cup E_1 \cup \{\epsilon\}, v_j, v_l \in V_2 \cup E_2 \cup \{\epsilon\}$, there is no edge between vertices in G_3 representing $v_i \to v_j$ and $v_k \to v_l$ if and only if $v_i = v_k \neq \epsilon$ or $v_j = v_l \neq \epsilon$.
  - For all $v_i \in V_1 \cup \{\epsilon\}, v_j \in V_2 \cup \{\epsilon\}, e(v_k, v_l) \in E_1, e(v_m, v_n) \in E_2$, there is no edge between vertices in G_3 representing $v_i \to v_j$ (any edit operation on vertices) and $e(v_k, v_l) \to e(v_m, v_n)$ (edge substitution) if:
    - $v_i = \epsilon$ and $v_j \in \{v_m, v_n\}$ (edge substitution is not compatible with insertion of an incident vertex),
    - $v_j = \epsilon$ and $v_i \in \{v_k, v_l\}$ (same for vertex deletion),
    - $v_i, v_j \neq \epsilon$ and ($v_i \in \{v_k, v_l\}$ and $v_j \notin \{v_m, v_n\}$ or $v_j \in \{v_m, v_n\}$ and $v_i \notin \{v_k, v_l\}$) (vertex substitution $v_i \to v_j$ is compatible with an edge substitution if and only if both v_i and v_j are mentioned in the edge substitution or neither).

    NB: there is always an edge between any edge insertion/deletion and any edit operation on vertices.
- Define L_{V_3} and $\mu_3 : V_3 \to L_{V_3}$ so that if $v_i \in V_3$ corresponds to any edit operation ed_i from G_1 to G_2, then $\mu_3(v_i) = c(ed_i)$.

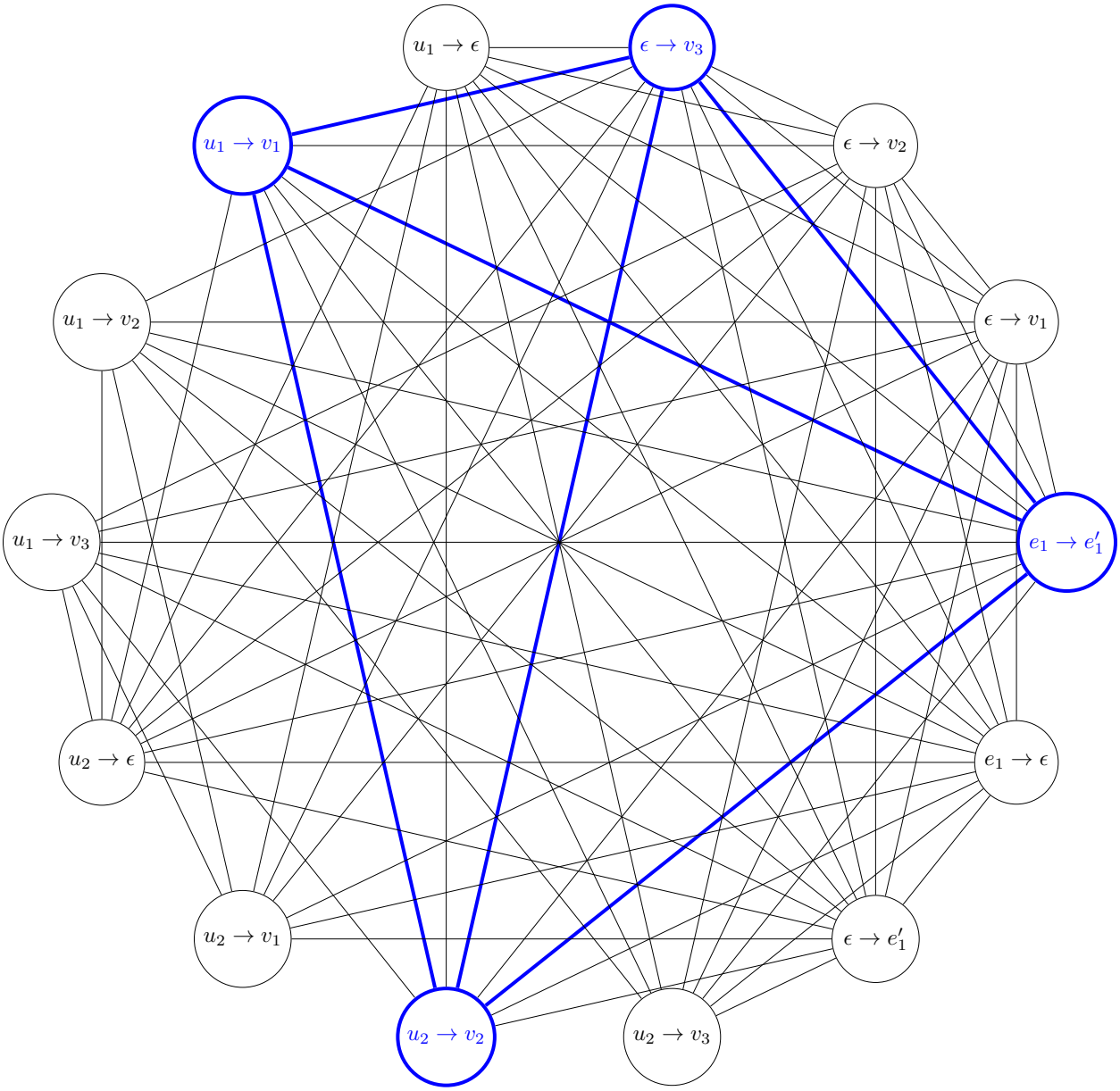

Figure 3: Vertex-weights encoding of the example from Figure 1 with the clique corresponding to the edit path $\star$ in blue

Then finding a minimum weight clique in G_3 is equivalent to finding the graph edit distance between G_1 and G_2, provided we add two new constraints to the MiniZinc model, ensuring that all edges in $E_1 \cup E_2$ have some edit operation associated with them. We will refer to this new model as the vertex-weights model. Figure 3 illustrates this encoding for graphs from Figure 1 with $e(u_1, u_2)$ and $e(v_1, v_2)$ denoted as e_1 and e'_1, respectively.

```
include "globals.mzn";

int: v1;
int: e1;
int: v2;
int: e2;
int: n = v1+v2+e1+e2+v1*v2+e1*e2; % number of vertices
int: e0 = (v1+1)*(v2+1) - 1; % index of the last vertex operation
array[1..n] of float: weights;
array[1..n,1..n] of 0..1: adjacent; % adjacency matrix
array[1..n] of var 0..1: clique; % whether a vertex is part of the clique
var float: distance;

% clique property
constraint forall(i in 1..n, j in 1..i-1)(
    adjacent[i,j] == 0 -> clique[i]+clique[j] <= 1);
% we must pick a vertex from each independent set
constraint forall(i in 1..v1)(sum(j in 0..v2)(clique[i*(v2+1)+j]) >= 1);
constraint forall(j in 1..v2)(sum(i in 0..v1)(clique[i*(v2+1)+j]) >= 1);
constraint forall(i in 1..e1)(sum(j in 0..e2)(clique[e0+i*(e2+1)+j]) >= 1);
constraint forall(j in 1..e2)(sum(i in 0..e1)(clique[e0+i*(e2+1)+j]) >= 1);
constraint distance == sum(i in 1..n)(clique[i] * weights[i]);

solve minimize distance;
```

2.4 The proposed algorithm

In order to use one of these clique encodings for any speed benefits, a state-of-the-art algorithm known as BITCLIQUE [6] (with its heuristic function called BITCOLOR, also known as Tavares' colouring) for solving maximum weight clique was adapted for the vertex-weights model. To find the graph edit distance between two attributed graphs $G_1 = (V_1, E_1, L_{V_1}, L_{E_1}, \mu_1, \zeta_1), G_2 = (V_2, E_2, L_{V_2}, L_{E_2}, \mu_2, \zeta_2)$, we create a new weighted graph $G = (V, E, L_V, \emptyset, \mu)$ according to the vertex-weights model, fix an order on $V_1 \cup V_2 \cup E_1 \cup E_2$, and construct a sequence of sets $(S_i)_{i=0}^{|V_1|+|V_2|+|E_1|+|E_2|-1}$, where $S_i \subset V$ is a set of all vertices corresponding to edit operations on the ith element of $V_1 \cup V_2 \cup E_1 \cup E_2$. These sets correspond to the additional constraints of the MiniZinc model. The resulting search algorithm is described in Algorithm 1 and the new heuristic function is described in Algorithm 2.

3 Implementation and evaluation

3.1 Data and its formatting

GDR4GED [2], a graph data repository for graph edit distance, was chosen to be used for algorithm evaluation and comparison. It contains four databases: GREC, MUTA, Protein, and CMU, with graph sizes ranging from 5 to 70 vertices. It also has ground truth values for some of the pairs of graphs, stating the optimal graph edit distance and a way to achieve it. To describe instances of this vertex-weights clique encoding, DIMACS format was extended in several ways:

Algorithm 1: The main algorithm with a recursive search function

Input: The number of vertices in the graph n, an adjacency matrix expressed as an array of bitsets *adjacencyBitsets*, a weight function $w : \{0, 1, ..., n-1\} \to \mathbb{R}^+$, a sequence of sets $(S_i)_{i=0}^{k-1}$

Output: A sequence of vertices $(v_j)_{j=1}^{k}$ comprising a clique in the graph such that for each set $S_i \in (S_i)_{i=0}^{k-1}$, there is a vertex $v_j \in (v_j)_{j=1}^{k}$ such that $v_j \in S_i$

bitset $P \leftarrow \underbrace{1 \dots 1}_{n}$;

initialise empty lists C, *incumbent*;

`expand`$(adjacencyBitsets, w, (S_i)_{i=0}^{k-1}, C, P, incumbent)$;

return *incumbent*;

Function `expand`$(adjacencyBitsets, w, (S_i)_{i=0}^{k-1}, C, P, incumbent)$

- initialise a 2-dimensional array *cumulativeWtBound* with k rows and $|S_i|$ columns for $0 \leq i < k$;
- $indSet \leftarrow$ `colouringBound`$(w, (S_i)_{i=0}^{k-1}, P, C, cumulativeWtBound)$;
- **if** $indSet = IMPOSSIBLE\_TO\_SATISFY$ **then**
  - **return**;
- **if** $indSet = ALL\_CONSTRAINTS\_SATISFIED$ **then**
  - **if** $|incumbent| = 0$ *or* $\sum_{v \in C} w(v) < \sum_{v \in incumbent} w(v)$ **then**
    - $incumbent \leftarrow C$;
  - **return**;
- **for** $v \in \{i : 0 \leq i < |S_{indSet}|,\ P[S_{indSet,i}] = 1\}$ *in the order of ascending* $cumulativeWtBound[indSet, v]$ **do**
  - C.`push`$(S_{indSet,v})$;
  - `expand`$(adjacencyBitsets, w, (S_i)_{i=0}^{k-1}, C, P \cap adjacencyBitsets[S_{indSet,v}], incumbent)$;
  - C.`pop`();

Algorithm 2: Tavares' colouring adapted to produce lower bounds for GED

Function `colouringBound`$(w, (S_i)_{i=0}^{k-1}, P, C, cumulativeWtBound)$

- $bound \leftarrow 0$;
- $smallestIndependentSet \leftarrow 0$;
- $minVerticesInP \leftarrow \infty$;
- $independentSetSatisfied = [\underbrace{0, 0, \ldots 0}_{k}]$;
- $residualWt \leftarrow [w(v) : 0 \leq v < n]$;
- `/* Choose the independent set with the smallest number of members in` P `to explore first */`
- **for** $i \leftarrow 0, \ldots, k-1$ **do**
  - $howManyVerticesInP \leftarrow 0$;
  - **for** $v \in S_i$ **do**
    - **if** $v \in C$ **then**
      - $independentSetSatisfied[i] = 1$;
      - **break**;
    - **if** $P[v] = 1$ **then**
      - $howManyVerticesInP \leftarrow howManyVerticesInP + 1$;
  - **if** $independentSetSatisfied[i] = 0$ **then**
    - **if** $howManyVerticesInP == 0$ **then**
      - **return** $IMPOSSIBLE\_TO\_SATISFY$; `// A negative integer constant`
    - **if** $howManyVerticesInP < minVerticesInP$ **then**
      - $smallestIndependentSet \leftarrow i$;
      - $minVerticesInP \leftarrow howManyVerticesInP$;
- **if** $minVerticesInP = \infty$ **then**
  - **return** $ALL\_CONSTRAINTS\_SATISFIED$; `// A different negative integer constant`
- **for** $i \leftarrow 0, \ldots, k-1$ **do**
  - **if** *(*$independentSetSatisfied[i] = 0$*) and* $i \neq smallestIndependentSet$ **then**
    - `colouringBoundForIndependentSet`$(i, residualWt, bound, P, cumulativeWtBound)$;
- `colouringBoundForIndependentSet`$(smallestIndependentSet, residualWt, bound, P, cumulativeWtBound)$;

Function `colouringBoundForIndependentSet`$(i, residualWt, bound, P, cumulativeWtBound)$

- $classMinWt \leftarrow \min\{residualWt[v] : v \in S_i, P[v] = 1\}$;
- **for** $j \leftarrow 0, \ldots, |S_i| - 1$ **do**
  - $residualWt[S_{i,j}] \leftarrow residualWt[S_{i,j}] - classMinWt$;
  - $cumulativeWtBound[i, j] \leftarrow bound + residualWt[S_{i,j}]$;

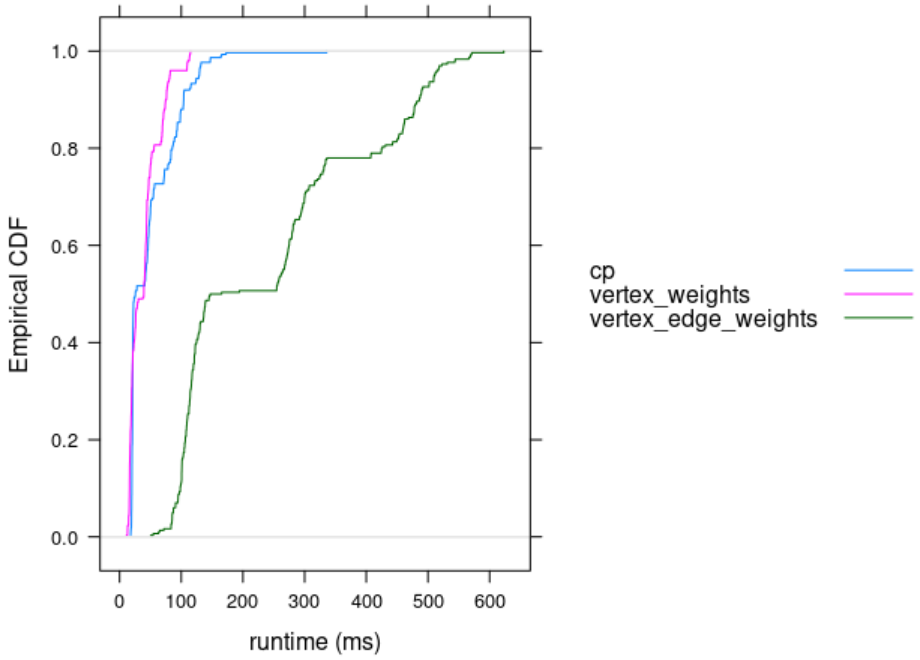

Figure 4: Comparing running times of all 3 models

- The first line now reads: `p edge n m v1 v2 e1 e2`, where:
  - `n` and `m` are the numbers of vertices and edges in the encoding graph, respectively (as usual),
  - `v1` and `v2` are the numbers of vertices in the original graphs,
  - `e1` and `e2` are the numbers of edges in the original graphs.

  The primary reason for this addition is that `v1` + `v2` + `e1` + `e2` is the number of independent sets that our model must pick a vertex from. Moreover, having all four numbers instead of just their sum allows us to recover information about which vertex represents which operation, which is not used by our main algorithm, but could be useful in deciding which independent set should be picked from first (variable ordering), which vertex is picked to satisfy the independent set (value ordering), or in considering different heuristic functions.

- Along with `e` indicating edges and `n` indicating vertex weights, we add lines starting with `s` indicating independent sets. Each such line is a space-separated list of vertex numbers (ranging from 1 to `n`), where we are required to pick one of them to be in the clique (picking more than one is always impossible since they form an independent set).

3.2 Evaluation

We begin by comparing the three MiniZinc models and plotting their cumulative (ECDF) plots in Figure 4, which shows the vertex-edge-weights model to be significantly inferior to the other two. The initial CP model and the vertex-weights encoding are about equal, with the latter eventually outperforming the former.

Our new algorithm (with its heuristic function) was implemented in C and proved to be orders of magnitude faster, finishing these 5-vertex GED problem instances with a maximum running time of 1 ms.

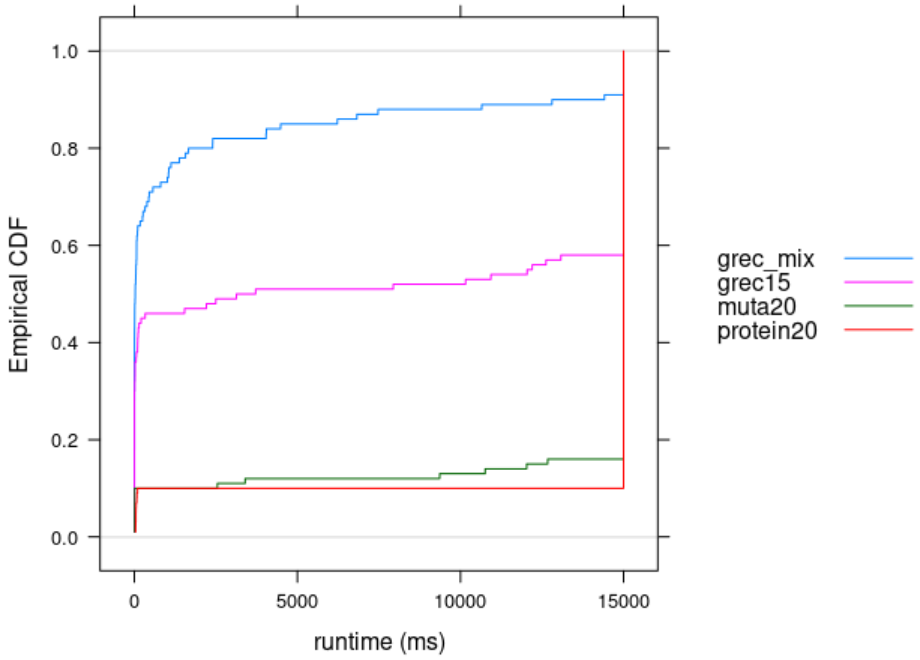

Figure 5: ECDF plots on various datasets with a 15 s time limit

Therefore, to evaluate it, we ran the algorithm with a 15 s time limit on GDR4GED datasets of appropriate sizes, namely: GRECMIX, GREC15, MUTA20, and Protein20. According to Figure 5, most problem instances are either very easy or very hard. The easy instances are usually the ones that have a very small optimal distance.

4 Discussion and related work

To compare our solution with other algorithms that are solving the same problem, we are only considering algorithms in the literature that are both exact and sequential. However, one approximate algorithm deserves special attention as it is the most widely used heuristic for many exact GED algorithms [3], including the ones reviewed in this section. The algorithm is known as the bipartite (BP) heuristic and works by reformulating graph matching as an assignment problem and solving it by using Munkres' algorithm [5]. There are two algorithms that we would like to highlight:

- A widely used algorithm for solving GED is based on A* search [4]. However, it suffers from high memory consumption [3].
- The state-of-the-art sequential algorithm for GED seems to be DF-GED [3], a branch-and-bound search algorithm.

Although we were not able to make direct comparisons by running these algorithms on the same machine, the hardware described in [3] is equivalent to the hardware used for our experiments. Thus, comparing our results on several GREC datasets, we can recognise that:

- DF-GED is able to find the optimal answer in 350 ms in all of the data (but not necessarily proves it to be correct), while Algorithm 1 has non-zero deviations (as defined in [3]) with a maximum of 0.05 with a 15 s time limit even on GREC15.

- A* fails to finish in 350 ms with most of GREC10 data, while Algorithm 1 successfully finishes with a mean running time of 34.73 ms and a maximum running time of 195 ms.

5 Conclusions and future work

Although the proposed algorithm did not outperform DF-GED, some important ideas were left untouched:

- Adapting the same algorithm to work with the vertex-edge-weights model. This might make the heuristic function less precise, but will result in less recursive function calls and handling quadratically less data.
- Exploring other heuristic functions, such as the aforementioned BP heuristic or one of its later variants.
- What vertex we choose to add to the clique at every node of the search tree can significantly affect the algorithm's performance. This is done in two stages:
  1. First we choose which independent set to pick from. We choose the set with the smallest number of vertices that can still be added to the clique.
  2. Then we order vertices in the selected independent set by their lower bounds and start with the smallest lower bound.

  Although these choices seem reasonable, perhaps there are better alternatives that should be explored.
- Creating a parallel version of the algorithm.

References

[1] Zeina Abu-Aisheh. "Anytime and Distributed Approaches for Graph Matching". PhD thesis. Université François-Rabelais de Tours, 2016.

[2] Zeina Abu-Aisheh, Romain Raveaux and Jean-Yves Ramel. "A Graph Database Repository and Performance Evaluation Metrics for Graph Edit Distance". In: *Graph-Based Representations in Pattern Recognition - 10th IAPR-TC-15 International Workshop, GbRPR 2015, Beijing, China, May 13-15, 2015. Proceedings*. Ed. by Cheng-Lin Liu et al. Vol. 9069. Lecture Notes in Computer Science. Springer, 2015, pp. 138–147. ISBN: 978-3-319-18223-0. DOI: 10.1007/978-3-319-18224-7_14. URL: https://doi.org/10.1007/978-3-319-18224-7_14.

[3] Zeina Abu-Aisheh et al. "An Exact Graph Edit Distance Algorithm for Solving Pattern Recognition Problems". In: *ICPRAM 2015 - Proceedings of the International Conference on Pattern Recognition Applications and Methods, Volume 1, Lisbon, Portugal, 10-12 January, 2015.* Ed. by Maria De Marsico, Mário A. T. Figueiredo and Ana L. N. Fred. SciTePress, 2015, pp. 271–278. ISBN: 978-989-758-076-5.

[4] Stefan Fankhauser, Kaspar Riesen and Horst Bunke. "Speeding Up Graph Edit Distance Computation through Fast Bipartite Matching". In: *Graph-Based Representations in Pattern Recognition - 8th IAPR-TC-15 International Workshop, GbRPR 2011, Münster, Germany, May 18-20, 2011. Proceedings*. Ed. by Xiaoyi Jiang, Miquel Ferrer and Andrea Torsello. Vol. 6658. Lecture Notes in Computer Science. Springer, 2011, pp. 102–111. ISBN: 978-3-642-20843-0. DOI: 10.1007/978-3-642-20844-7_11. URL: https://doi.org/10.1007/978-3-642-20844-7_11.

[5] Kaspar Riesen and Horst Bunke. "Approximate graph edit distance computation by means of bipartite graph matching". In: *Image Vision Comput.* 27.7 (2009), pp. 950–959. DOI: 10.1016/j.imavis.2008.04.004. URL: https://doi.org/10.1016/j.imavis.2008.04.004.

[6] Wladimir Araujo Tavares. "Algoritmos exatos para problema da clique maxima ponderada. (Exact algorithms for the maximum-weight clique problem / Algorithmes pour le problème de la clique de poids maximum)". PhD thesis. University of Avignon, France, 2016. URL: https://tel.archives-ouvertes.fr/tel-01401999.

[7] Philippe Vismara and Benoît Valery. "Finding Maximum Common Connected Subgraphs Using Clique Detection or Constraint Satisfaction Algorithms". In: *Modelling, Computation and Optimization in Information Systems and Management Sciences, Second International Conference, MCO 2008, Metz, France - Luxembourg, September 8-10, 2008. Proceedings*. Ed. by Le Thi Hoai An, Pascal Bouvry and Pham Dinh Tao. Vol. 14. Communications in Computer and Information Science. Springer, 2008, pp. 358–368. ISBN: 978-3-540-87476-8. DOI: 10.1007/978-3-540-87477-5_39. URL: https://doi.org/10.1007/978-3-540-87477-5_39.