Flexible Graph Matching and Graph Edit Distance Using Answer Set Programming

Sheung Chi Chan[1] and James Cheney[1,2](✉)

[1] University of Edinburgh, Edinburgh, UK
jcheney@inf.ed.ac.uk
[2] The Alan Turing Institute, London, UK

Abstract. The *graph isomorphism*, *subgraph isomorphism*, and *graph edit distance* problems are combinatorial problems with many applications. Heuristic exact and approximate algorithms for each of these problems have been developed for different kinds of graphs: directed, undirected, labeled, etc. However, additional work is often needed to adapt such algorithms to different classes of graphs, for example to accommodate both labels and property annotations on nodes and edges. In this paper, we propose an approach based on answer set programming. We show how each of these problems can be defined for a general class of *property graphs* with directed edges, and labels and key-value properties annotating both nodes and edges. We evaluate this approach on a variety of synthetic and realistic graphs, demonstrating that it is feasible as a rapid prototyping approach.

1 Introduction

Graphs are a pervasive and widely applicable data structure in computer science. To name just a few examples, graphs can represent symbolic knowledge structures extracted from Wikipedia [5], provenance records describing how a computer system executed to produce a result [20], or chemical structures in a scientific knowledge base [15]. In many settings, it is of interest to solve *graph matching* problems, for example to determine when two graphs have the same structure, or when one graph appears in another, or to measure how similar two graphs are.

Given two graphs, possibly with labels or other data associated with nodes and edges, the *graph isomorphism* problem (GI) asks whether the two graphs have the same structure, that is, whether there is an invertible mapping from one graph to another that preserves and reflects edges and any other constraints. The *subgraph isomorphism* problem (SUB) asks whether one graph is isomorphic to a subgraph of another. Finally, the *graph edit distance* problem (GED) asks whether one graph can be transformed into another via a sequence of edit steps, such as insertion, deletion, or updates to nodes or edges.

These are well-studied problems. Each is in the class NP, with SUB and GED being NP-complete [12], while the lower bound of the complexity of GI is an open

© Springer Nature Switzerland AG 2020
E. Komendantskaya and Y. A. Liu (Eds.): PADL 2020, LNCS 12007, pp. 20–36, 2020.
https://doi.org/10.1007/978-3-030-39197-3_2

problem [4]. Approximate and exact algorithms for graph edit distance, based on heuristics or on reduction to other NP-complete problems, have been proposed [9,11,17,21]. Moreover, for special cases such as database querying, there are algorithms for subgraph isomorphism that can provide good performance in practice when matching small query subgraphs against graph databases [16].

However, there are circumstances in which none of the available techniques is directly suitable. For example, many of the algorithms considered so far assume graphs of a specific form, for example with unordered edges, or unlabeled nodes and edges. In contrast, many typical applications use graphs with complex structure, such as property graphs: directed multigraphs in which nodes and edges can both be labeled and annotated with sets of key-value pairs (*properties*). Adapting an existing algorithm to deal with each new kind of graph is nontrivial. Furthermore, some applications involve searching for isomorphisms, subgraph isomorphisms, or edit scripts subject to additional constraints [8,22].

In this paper we advocate the use of *answer set programming* (ASP) to specify and solve these problems. Property graphs can be represented uniformly as sets of logic programming facts, and each of the graph matching problems we have mentioned can be specified using ASP in a uniform way. Concretely, we employ the Clingo ASP solver, but our approach relies only on standard ASP features.

For each of the problems we consider, it is clear in principle that it should be possible to encode using ASP, because ASP subsumes the NP-complete SAT problem. Our contribution is to show how to encode each of these problems directly in a way that produces immediately useful results, rather than via encoding as SAT or other problems and decoding the results. For GI and SUB, the encoding is rather direct and the ASP specifications can easily be read as declarative specifications of the respective problems; however, the standard formulation of the graph edit distance problem is not as easy to translate to a logic program because it involves searching for an edit script whose maximum length depends on the input. Instead, we consider an indirect (but still natural) approach which searches for a partial matching between the two graphs that minimizes the edit distance, and derives an edit script (if needed) from this matching. The proof of correctness of this encoding is our main technical contribution.

We provide experimental evidence of the practicality of our declarative approach, drawing on experience with a nontrivial application: generalizing and comparing provenance graphs [8]. In this previous work, we needed to solve two problems: (1) given two graphs with the same structure but possibly different property values (e.g. timestamps), identify the general structure common to all of the graphs, and (2) given a background graph and a slightly larger foreground graph, match the background graph to the foreground graph and “subtract” it, leaving the unmatched part. We showed in [8] that our ASP approach to approximate graph isomorphism and subgraph isomorphism can solve these problems fast enough that they were not the bottleneck in the overall system. In this paper, we conduct further experimental evaluation of our approach to graph isomorphism, subgraph isomorphism, and graph edit distance on synthetic graphs

and real graphs used in a recent Graph Edit Distance Contest (GEDC) [1] and our recent work [8].

2 Background

Property Graphs. We consider *(directed) multigraphs* $G = (V, E, src, tgt, lab)$ where V and E are disjoint sets of *node identifiers* and *edge identifiers*, respectively, $src, tgt : E \rightarrow V$ are functions identifying the source and target of each edge, and $lab : V \cup E \rightarrow \Sigma$ is a function assigning each vertex and edge a label from some set Σ. Note that multigraphs can have multiple edges with the same source and target. Familiar definitions of ordinary directed or undirected graphs can be recovered by imposing further constraints, if desired.

A *property graph* is a directed multigraph extended with an additional partial function $prop : (V \cup E) \times \Gamma \rightharpoonup \Delta$ where Γ is a set of *keys* and Δ is a set of *data values*. For the purposes of this paper we assume that all identifiers, labels, keys and values are represented as Prolog atoms.

We consider a partial function with range X to be a total function with range $X \uplus \{\bot\}$ where $\bot$ is a special token not appearing in X. We consider $X \uplus \{\bot\}$ to be partially ordered by the least partial order satisfying $\bot \sqsubseteq x$ for all $x \in X$.

Isomorphisms. A *homomorphism* from property graph G_1 to G_2 is a function $h : G_1 \rightarrow G_2$ mapping V_1 to V_2 and E_1 to E_2, such that:

- for all $v \in V_1$, $lab_2(h(v)) = lab_1(v)$ and $prop_2(h(v), k) \sqsubseteq prop_1(v, k)$
- for all $e \in E_1$, $lab_2(h(e)) = lab_1(e)$ and $prop_2(h(e), k) \sqsubseteq prop_1(e, k)$
- for all $e \in E_1$, $src_2(h(e)) = h(src_1(e))$ and $tgt_2(h(e)) = h(tgt_1(e))$

(Essentially, h is a pair of functions $(V_1 \rightarrow V_2) \times (E_1 \rightarrow E_2)$, but we abuse notation slightly here by writing h for both.) As usual, an isomorphism is an invertible homomorphism whose inverse is also a homomorphism, and G_1 and G_2 are isomorphic ($G_1 \cong G_2$) if an isomorphism between them exists. Note that the labels of nodes and edges must match exactly, that is, we regard labels as integral to nodes and edges, while properties must match only if defined in G_1.

Subgraph Isomorphism. A subgraph G' of G is a property graph satisfying:

- $V' \subseteq V$ and $E' \subseteq E$
- $src'(e) = src(e) \in V'$ and $tgt(e) = tgt'(e) \in V'$ for all $e \in E'$
- $lab'(x) = lab(x)$ when $x \in V' \cup E'$
- $prop'(x, k) \sqsubseteq prop(x, k)$ when $x \in V' \cup E'$

In other words, the vertex and edge sets of G' are subsets of those of G that still form a meaningful graph, the labels are the same as in G', and the properties defined in G' are the same as in G (but some properties in G may be omitted).

We say that G_1 is *subgraph isomorphic* to G_2 ($G_1 \lesssim G_2$) if there is a subgraph of G_2 to which G_1 is isomorphic. Equivalently, $G_1 \lesssim G_2$ holds if there is a *injective* homomorphism $h : G_1 \rightarrow G_2$. If such a homomorphism exists, then it maps G_1 to an isomorphic subgraph of G_2, whereas if $G_1 \cong G_2' \subseteq G_2$ then the isomorphism between G_1 and G_2' extends to an injective homomorphism from G_1 to G_2.

Table 1. Edit operation semantics

op	V'	E'	src'	tgt'	lbl'	$prop'$
$\mathsf{insV}(n, l)$	$V \uplus \{v\}$	E	src	tgt	$lbl[v := l]$	$prop$
$\mathsf{insE}(e, v, w, l)$	V	$E \uplus \{e\}$	$src[e := v]$	$tgt[e := w]$	$lbl[e := l]$	$prop$
$\mathsf{insP}(x, k, d)$	V	E	src	tgt	lbl	$prop[x, k := d]$
$\mathsf{delV}(v)$	$V - \{v\}$	E	src	tgt	$lbl[v := \bot]$	$prop$
$\mathsf{delE}(e)$	V	$E - \{e\}$	$src[e := \bot]$	$tgt[e := \bot]$	$lbl[e := \bot]$	$prop$
$\mathsf{delP}(x, k)$	V	E	src	tgt	lbl	$prop[x, k := \bot]$
$\mathsf{updP}(x, k, d)$	V	E	src	tgt	lbl	$prop[x, k := d]$

Graph Edit Distance. We consider *edit operations*:

- insertion of a node ($\mathsf{insV}(v, l)$), edge ($\mathsf{insE}(e, v, w, l)$), or property ($\mathsf{insP}(x, k, v, d)$)
- deletion of a node ($\mathsf{delV}(v)$), edge ($\mathsf{delE}(e)$), or property ($\mathsf{delP}(x, k)$)
- in-place update ($\mathsf{updP}(x, k, d)$) of a property value on a given node or edge x with a given key k to value d

The meanings of each of these operations are defined in Table 1, where we write $G = (V, E, src, tgt, lab, prop)$ for the graph before the edit and $G' = (V', E', src', tgt', lab', prop')$ for the updated graph. Each row of the table describes how each part of G' is defined in terms of G. In addition, the edit operations have the following preconditions: Before an insertion, the inserted node, edge, or property must not already exist; before a deletion, a deleted node must not be a source or target of an edge, and a node/edge must not have any properties; before an update, the updated property must already exist on the affected node or edge. If these preconditions are not satisfied, the edit operation is not allowed on G.

We write $op(G)$ for the result of op acting on G. More generally, if ops is a list of operations then we write $ops(G)$ for the result of applying the operations to G. Given graphs G_1, G_2 we define the *graph edit distance* between G_1 and G_2 as $GED(G_1, G_2) = \min\{|ops| \mid ops(G_1) = G_2\}$, that is, the shortest length of an edit script modifying G_1 to G_2.

Computing the graph edit distance between two graphs (even without labels or properties) is an NP-complete problem. Moreover, we consider a particular setting where the edit operations all have equal cost, but in general different weights can be assigned to different edit operations. We can consider a slight generalization as follows: Given a weighting function w mapping edit operations to positive rational numbers, the *weighted graph edit distance* between G_1 and G_2 is $wGED(G_1, G_2) = \min\{\sum_{op \in ops} w(op) \mid ops(G_1) = G_2\}$. The unweighted graph edit distance is a special case so this problem is also NP-complete.

Answer Set Programming. We assume familiarity with general logic programming concepts (e.g. familiarity with Prolog or Datalog). To help make the paper accessible to readers not already familiar with answer set programming, we illustrate some programming techniques that differ from standard logic programming

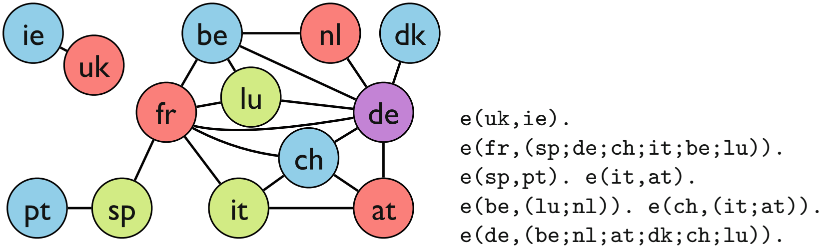

Fig. 1. Graph coloring example (Color figure online)

via a short example: coloring the nodes of an undirected graph with the minimum number of colors. Graph 3-coloring is a standard example of ASP, but we will adopt a slightly nonstandard approach to illustrate some key techniques we will rely on later. We will use the concrete syntax of the Clingo ASP solver, which is part of the Potassco framework [13,14]. Examples given here and elsewhere in the paper can be run verbatim using the Clingo interactive online demo[1].

Listing 1.1. Graph 3-coloring

```
1 e(X,Y) :- e(Y,X).
2 n(X) :- e(X,_).
3 color(1..3).
4 {c(X,Y) : color(Y)} = 1 :- n(X).
5 :- e(X,Y), c(X,C), c(Y,D), not C <> D.
```

Listing 1.2. Minimal k-coloring (extending Listing 1.1)

```
1 color(X) :- n(X).
2 cost(C,1) :- c(_,C).
3 #minimize { Cost,C : cost(C,Cost) }.
```

Figure 1 shows an example graph where edge relationships correspond to land borders between some countries. The edges are defined using an association list notation; for example `e(be,(lu;nl))` abbreviates two edges `e(be,lu)` and `e(be,nl)`. Listing 1.1 defines graph 3-coloring declaratively. The first line states that the edge relation is symmetric and the second defines the node relation to consist of all sources (and by symmetry targets) of edges. Line 3 defines a relation `color/1` to hold for values 1, 2, 3. Lines 4–5 define when a graph is 3-colorable, by defining when a relation `c/2` is a valid 3-coloring. Line 4 says that `c/2` represents a (total) function from nodes to colors, i.e. for every node there is exactly one associated color. Line 5 says that for each edge, the associated colors of the source and target must be different. Here, we are using the `not` operator solely to illustrate its use, but we could have done without it, writing `C = D` instead.

[1] https://potassco.org/clingo/run/.

Listing 1.1 is a complete program that can be used with Fig. 1 to determine that the example graph is not 3-colorable. What if we want to find the least k such that a graph is k-colorable? We cannot leave the number of colors undefined, since ASP requires a finite search space, but we could manually change the '3' on line 5 to various values of k, starting with the maximum $k = |V|$ and decreasing until the minimum possible k is found.

Instead, using *minimization constraints*, we can modify the 3-coloring program above to instead compute a minimal k-coloring (that is, find a coloring minimizing the number of colors) purely declaratively by adding the clauses shown in Listing 1.2. Line 1 defines the set of colors simply to be the set of node identifiers (plus the three colors we already had, but this is harmless). Line 2 associates a cost of 1 with each used color. Finally, line 3 imposes a minimization constraint: to minimize the sum of the costs of the colors. Thus, using a single Clingo specification we can automatically find the minimum number of colors needed for this (or any) undirected graph. The 4-coloring shown in Fig. 1 was found this way.

3 Specifying Graph Matching and Edit Distance

In this section we give ASP specifications defining each problem. We first consider how to represent graphs as flat collections of facts, suitable for use in a logic programming setting. We choose one among several reasonable representations: given $G = (V, E, \mathit{src}, \mathit{tgt}, \mathit{lab}, \mathit{prop})$ and given three predicate names $\mathtt{n}, \mathtt{e}, \mathtt{p}$ we define the following relations:

$$\begin{aligned} \mathit{Rel}_G(\mathtt{n}, \mathtt{e}, \mathtt{p}) &= \{\mathtt{n}(v, \mathit{lab}(v)) \mid v \in V\} \\ &\cup \{\mathtt{e}(e, \mathit{src}(e), \mathit{tgt}(e), \mathit{lab}(e)) \mid e \in E\} \\ &\cup \{\mathtt{p}(x, k, d) \mid x \in V \cup E, \mathit{prop}(x, k) = d \neq \bot\} \end{aligned}$$

Clearly, we can recover the original graph from this representation.

In the following problem specifications, we always consider two graphs, say G_1 and G_2, and to avoid confusion between them we use two sets of relation names to encode them, thus $\mathit{Rel}_{G_1}(\mathtt{n}_1, \mathtt{e}_1, \mathtt{p}_1) \cup \mathit{Rel}_{G_2}(\mathtt{n}_2, \mathtt{e}_2, \mathtt{p}_2)$ represents two graphs. We also assume without loss of generality that the sets of vertex and edge identifiers of the two graphs are all disjoint, i.e. $(V_1 \cup E_1) \cap (V_2 \cup E_2) = \emptyset$, to avoid any possibility of confusion among them.

We now show how to specify homomorphisms and isomorphisms among graphs. The Clingo code in Listing 1.3 defines when a graph homomorphism exists from G_1 to G_2. We refer to this program extended with suitable representations of G_1 and G_2 as $\mathit{Hom}_h(G_1, G_2)$. The binary relation h, representing the homomorphism, is specified using two constraints. The first says that h maps nodes of G_1 to nodes of G_2 with the same label, while the second additionally specifies that h maps edges of G_1 to those of G_2 preserving source, target, and label. Notice in particular that the cardinality constraint ensures that h represents a total function with range $V_1 \cup E_1$, so in any model satisfying the first

clause, every node in G_1 is matched to one in G_2, which means that the body of the second clause is satisfiable for each edge. The third clause simply constrains h so that any properties of nodes or edges in G_1 must be present on the matching node or edge in G_2.

Listing 1.3. Graph homomorphism

```
1 {h(X,Y) : n2(Y,L)} = 1 :- n1(X,L).
2 {h(X,Y) : e2(Y,S2,T2,L), h(S1,S2), h(T1,T2)} = 1 :- e1(X,S1,T1,L).
3 :- p1(X,K,D), h(X,Y), not p2(Y,K,D).
```

Listing 1.4. Graph isomorphism (extending Listing 1.3)

```
1 {h(X,Y) : n1(X,L)} = 1 :- n2(Y,L).
2 {h(X,Y) : e1(X,S1,T1,L), h(S1,S2), h(T1,T2)} = 1 :- e2(Y,S2,T2,L).
3 :- p2(Y,K,D), h(X,Y), not p1(X,K,D).
```

Listing 1.5. Subgraph isomorphism (extending Listing 1.3)

```
1 {h(X,Y) : n1(X,L)} <= 1 :- n2(Y,L).
2 {h(X,Y) : e1(X,S1,T1,L), h(S1,S2), h(T1,T2)} <= 1 :- e2(Y,S2,T2,L).
```

Next to define when h is a graph *isomorphism*, we add the symmetric clauses shown in Listing 1.4. We write $Iso_h(G_1, G_2)$ for the combination of Listings 1.3 and 1.4. Since the two listings together imply that h represents a homomorphism in the forward direction and simultaneously represents a homomorphism from G_2 to G_1 in the backward direction, these four clauses suffice to specify that h is an isomorphism.

To specify subgraph isomorphism, we simply require that h is an injective homomorphism from G_1 to G_2, as shown in Listing 1.5. We refer to the specification in Listing 1.5 as $Sub_h(G_1, G_2)$. The two additional constraints specify that the inverse of h is a *partial* homomorphism. This is equivalent to h being an injective homomorphism.

Finally we consider the specification of the graph edit distance problem. On the surface, this seems challenging, since the graph edit distance is defined as the length of a minimal edit script mapping one graph to another, and there are infinitely many possible edit scripts. However, there is clearly always an upper bound d on the edit distance: consider an edit script that just deletes G_1 and inserts G_2, and take d to be the length of this script. So, given two graphs and this upper bound d we could proceed by specifying a search space over edit scripts of bounded length, defining the meaning of each edit operator, and seeking to minimize the number of steps necessary to get from G_1 to G_2. However, this encoding seems rather heavyweight, and requires preprocessing to determine d.

Instead, we follow a different strategy, analogous to the approach adopted for graph coloring earlier. The strategy is based on the observation that the graph edit distance is closely related to the *maximum subgraph problem* [6], that is, given two graphs G_1, G_2, find the largest graph that is subgraph isomorphic to

both. If we identify such a graph then (as we shall show) we can read off an edit script that maps G_1 to G_2, which first deletes unmatched structure from G_1, then updates properties in-place, and finally inserts new structure needed in G_2. Furthermore, to identify the maximum common subgraph, we do not need to construct a new graph separate from G_1 and G_2; instead, we can think of the maximum common subgraph as an isomorphic pair of subgraphs of G_1 and G_2. So in other words, we will search for a partial isomorphism h between G_1 and G_2, use it as a basis for extracting an edit script, and minimize its cost.

Listing 1.6. Graph edit distance

```
1  {h(X,Y) : n2(Y,L)} <= 1 :- n1(X,L).
2  {h(X,Y) : n1(X,L)} <= 1 :- n2(Y,L).
3  {h(X,Y) : e2(Y,S2,T2,L), h(S1,S2), h(T1,T2)} <= 1 :- e1(X,S1,T1,L).
4  {h(X,Y) : e1(X,S1,T1,L), h(S1,S2), h(T1,T2)} <= 1 :- e2(Y,S2,T2,L).
5
6  delete_node(X) :- n1(X,_), not h(X,_).
7  insert_node(Y,L) :- n2(Y,L), not h(_,Y).
8
9  delete_edge(X) :- e1(X,_,_,_), not h(X,_).
10 insert_edge(Y,S,T,L) :- e2(Y,S,T,L), not h(_,Y).
11
12 update_prop(X,K,V1,V2) :- p1(X,K,V1), h(X,Y), p2(Y,K,V2), V1 <> V2.
13 delete_prop(X,K) :- p1(X,K,_), h(X,Y), not p2(Y,K,_).
14 delete_prop(X,K) :- p1(X,K,_), delete_node(X).
15 delete_prop(X,K) :- p1(X,K,_), delete_edge(X).
16 insert_prop(Y,K,V) :- p2(Y,K,V), h(X,Y), not p1(X,K,_).
17 insert_prop(Y,K,V) :- p2(Y,K,V), insert_node(Y,_).
18 insert_prop(Y,K,V) :- p2(Y,K,V), insert_edge(Y,_,_,_).
19
20 node_cost(Y,1) :- insert_node(Y,_).
21 node_cost(X,1) :- delete_node(X).
22
23 edge_cost(Y,1) :- insert_edge(Y,_,_,_).
24 edge_cost(X,1) :- delete_edge(X).
25
26 prop_cost(X,K,1) :- update_prop(X,K,V1,V2).
27 prop_cost(X,K,1) :- delete_prop(X,K).
28 prop_cost(Y,K,1) :- insert_prop(Y,K,V).
29
30 #minimize { NC,X : node_cost(X,NC);
31             EC,X : edge_cost(X,EC);
32             LC,X,K : prop_cost(X,K,LC)}.
```

Listing 1.6 accomplishes this. The first four lines specify that h must be a partial isomorphism, by dropping the requirement that h must match all nodes/edges on one side with those of another, and dropping the hard constraint that properties must match. Lines 6–7 define when a node must be deleted or inserted. Nodes that are in G_1 and not matched in G_2 must be deleted, and

conversely those that are in G_2 and not matched in G_1 must be inserted. Lines 9–10 similarly specify when edges must be inserted or deleted. Lines 12–18 define when a property is updated in-place, deleted, or inserted. If a property key is present on an object in G_1 and on the matching object in G_2 but with a different value, then the key's value needs to be updated. If it is present in G_1 but not present on the matching object in G_1 then it is deleted. Likewise, if it is present in G_1 but the associated object is deleted then the property also must be deleted. Dually, properties are inserted if they are present in G_2 but not in G_1, either because the matching object does not have that property or because there is no matching object because the property is on an inserted object. Lines 20–28 specify the costs associated with each of the edit operations. We assign each operation a cost of 1. It would also be possible to assign different (integer) costs to different kinds of updates, or even to specify different costs depending on labels, keys, or values.

4 Correctness

We first state the intended correctness properties for the homomorphism, isomorphism, and subgraph isomorphism problems:

Theorem 1. *1. There exists a homomorphism $h : G_1 \to G_2$ if and only if $Hom_h(G_1, G_2)$ is satisfiable.*
2. There exists an isomorphism $h : G_1 \to G_2$ if and only if $Iso_h(G_1, G_2)$ is satisfiable.
3. $h : G_1 \to G_2$ witnesses a subgraph isomorphism if and only if $Sub_h(G_1, G_2)$ is satisfiable.

Proof. See Appendix A of the extended version [7]. □

Next we turn to graph edit distance. To assist with the reasoning, we define the following canonical form:

Definition 1 (Edit script canonical form). *An edit script is in* canonical form *if it is of the form $del_p; del_e; del_v; upd_p; ins_v; ins_e; ins_p$, where:*

- *del_p, del_e and del_v are sequences of property deletions, edge deletions, and node deletions respectively;*
- *upd_p is a sequence of property updates;*
- *ins_v, ins_e, and ins_p are sequences of node insertions, edge insertions, and property insertions, respectively.*

Edit scripts obtained from $GED_h(G_1, G_2)$ are in this form. Moreover, any valid edit script can be converted to a canonical one by applying a set of rewrite rules, as shown in Fig. 2. We first consider *marked* versions op^* of each edit operation, for example writing $\mathsf{delP}^*(x, k)$ for the marked version of delP. A marked operation op^* has the same effect as op when applied to a graphs; the mark is only to indicate which operation is actively being rewritten. The idea here is that if we

$$
\begin{array}{rcl}
\mathsf{delE}^*(e); \mathsf{delP}(x,k) & \longrightarrow & \mathsf{delP}(x,k); \mathsf{delE}^*(e) \\
\mathsf{delV}^*(v); \mathsf{delP}(x,k) & \longrightarrow & \mathsf{delP}(x,k); \mathsf{delV}^*(v) \\
\mathsf{delV}^*(v); \mathsf{delE}(e) & \longrightarrow & \mathsf{delE}(e); \mathsf{delV}^*(v) \\
\mathsf{updP}^*(x,k,d); \mathsf{delP}(y,k') & \longrightarrow & \begin{cases} \mathsf{delP}(y,k') & \text{if } x = y, k = k' \\ \mathsf{delP}(y,k'); \mathsf{updP}^*(x,k,d) & \text{otherwise} \end{cases} \\
\mathsf{updP}^*(x,k,d); \mathsf{delE}(e) & \longrightarrow & \mathsf{delE}(e); \mathsf{updP}^*(x,k,d) \\
\mathsf{updP}^*(x,k,d); \mathsf{delV}(v) & \longrightarrow & \mathsf{delV}(v); \mathsf{updP}^*(x,k,d) \\
\mathsf{insV}^*(v,l); \mathsf{delP}(x,k) & \longrightarrow & \mathsf{delP}(x,k); \mathsf{insV}^*(v,l) \\
\mathsf{insV}^*(v,l); \mathsf{delE}(e) & \longrightarrow & \mathsf{delE}(e); \mathsf{insV}^*(v) \\
\mathsf{insV}^*(v,l); \mathsf{delV}(v') & \longrightarrow & \begin{cases} \epsilon & \text{if } v = v' \\ \mathsf{delV}(v'); \mathsf{insV}^*(v,l) & \text{otherwise} \end{cases} \\
\mathsf{insV}^*(v,l); \mathsf{updP}(x,k,d) & \longrightarrow & \mathsf{updP}(x,k,d); \mathsf{insV}^*(v,l) \\
\mathsf{insE}^*(e,v,w,l); \mathsf{delP}(x,k) & \longrightarrow & \mathsf{delP}(x,k); \mathsf{insE}^*(e,v,w,l) \\
\mathsf{insE}^*(e,v,w,l); \mathsf{delE}(e') & \longrightarrow & \begin{cases} \epsilon & \text{if } e = e' \\ \mathsf{delE}(e'); \mathsf{insE}^*(e,v,w,l) & \text{otherwise} \end{cases} \\
\mathsf{insE}^*(e,v,w,l); \mathsf{delV}(v') & \longrightarrow & \mathsf{delV}(v'); \mathsf{insE}^*(e,v,w,l) \\
\mathsf{insE}^*(e,v,w,l); \mathsf{updP}(x,k,d) & \longrightarrow & \mathsf{updP}(x,k,d); \mathsf{insE}^*(e,v,w,l) \\
\mathsf{insE}^*(e,v,w,l); \mathsf{insV}(v',l) & \longrightarrow & \mathsf{insV}(v',l); \mathsf{insE}^*(e,v,w,l) \\
\mathsf{insP}^*(x,k,d); \mathsf{delP}(y,k') & \longrightarrow & \begin{cases} \epsilon & \text{if } x = y, k = k' \\ \mathsf{delP}(y,k'); \mathsf{insP}^*(x,k,d) & \text{otherwise} \end{cases} \\
\mathsf{insP}^*(x,k,d); \mathsf{delE}(e) & \longrightarrow & \mathsf{delE}(e); \mathsf{insP}^*(x,k,d) \\
\mathsf{insP}^*(x,k,d); \mathsf{delV}(v) & \longrightarrow & \mathsf{delV}(v); \mathsf{insP}^*(x,k,d) \\
\mathsf{insP}^*(x,k,d); \mathsf{updP}(y,k',d') & \longrightarrow & \begin{cases} \mathsf{insP}^*(x,y,d') & \text{if } x = y, k = k' \\ \mathsf{updP}(y,k',d'); \mathsf{insP}^*(x,k,d) & \text{otherwise} \end{cases} \\
\mathsf{insP}^*(x,k,d); \mathsf{insV}(v,l) & \longrightarrow & \mathsf{insV}(v',l); \mathsf{insP}^*(x,k,d) \\
\mathsf{insP}^*(x,k,d); \mathsf{insE}(e,v,w,l) & \longrightarrow & \mathsf{insE}(e,v,w,l); \mathsf{insP}^*(x,k,d) \\
op^*; ops & \longrightarrow & op; ops \qquad \text{if no earlier rule applies}
\end{array}
$$

Fig. 2. Edit script rewrite rules

have a canonical edit script *ops* and wish to add a new edit operation, we use the rewrite rules to canonicalize $op^*; ops$. The rules are applied in order and at each step, the first matching rule is applied. Note that there is a catch-all rule $op^*; ops \longrightarrow op; ops$, which only applies if none of the other rules do. Essentially, the rewrite rules consider all of the possible pairs of adjacent operations that can appear in a non-canonical form, with the first element marked. In each case, they show how to simplify the edit script by either moving the marked operation closer to the end, or removing the mark. Removal can happen as a result of either cancellation of the marked operation by another operation (e.g. a delete undoing an insert), or by removing the mark once it has reached an appropriate place for it in the canonical form.

Lemma 1. *If ops is an edit script mapping G_1 to G_2, then there is a canonical edit script ops' mapping G_1 to G_2 such that $|ops'| \leq |ops|$.*

Proof. See Appendix A of the extended version [7]. □

Theorem 2. *The specification* $GED_h(G_1, G_2)$ *always has a solution, and the edit script described by the insertion, deletion and update predicates is a valid, canonical script mapping* G_1 *to* G_2*. Moreover, the cost of the optimal solution to* $GED_h(G_1, G_2)$ *equals* $GED(G_1, G_2)$*.*

Proof. For the first part, we observe that the empty relation $h = \emptyset$ always solves $GED_h(G_1, G_2)$ if we ignore the minimization constraint. Therefore, the cost of this solution is an upper bound. Moreover, if we apply the edit operations described by the insert, delete and update relations in the order required by the canonical form, then each edit operation is valid, all structure present in G_1 and not G_2 is removed, all properties whose values differ in G_1 and G_2 are updated, and all structure present in G_2 and not G_1 is inserted. Therefore, the corresponding edit script maps G_1 to G_2.

To show that the minimum cost obtained from solving the $GED_h(G_1, G_2)$ specification coincides with $GED(G_1, G_2)$, one direction is easy: for any h (including the one corresponding to a minimum cost solution) the collection of edit operations resulting from $GED_h(G_1, G_2)$ is a valid edit script so its length d must be greater than or equal to the minimum over all valid scripts. To show the reverse direction, we use Lemma 1. Given a minimum-length edit script that is not in canonical form, we can rewrite it to one that is canonical, with equal cost (since the original script was already minimum-length). □

5 Discussion

We have argued that using ASP offers considerable flexibility. To illustrate this claim, we consider three modifications to our approach.

Weighted Graph Edit Distance. If the operations have different (integer) weights, implemented using a suitable modification to the cost predicates in some specification $wGED_h(G_1, G_2)$, then the same argument as above suffices to show that a minimum-weight canonical script always exists to be found by the ASP specification. The key point is that weights are defined on individual edit operations, and the rewrite rules only permute or delete operations, so preserve or decrease weight.

Relabeling. We have treated labels as hard constraints: it is not possible to change the label of a node in G_1 to a different label in G_2, short of deleting the node and inserting a new one with a different label. On the other hand, properties are soft constraints in the sense that we may delete or update a property value without also being obliged to delete and re-create the underlying node or edge structure. It is natural to consider an in-place relabeling operation as well. Such behavior can be encoded on top of the already-developed framework by using a single "blank" label for nodes and edges and introducing an unused property key called "label" instead; now this can be updated in-place like other properties. Alternatively, we can accommodate this behavior more directly as shown in Listing 1.7. The first four lines relax the constraint that node and edge

labels have to be preserved by h. The next two lines define the `relabel_node` and `relabel_edge` predicates to detect when two matched nodes or edges have different labels. Finally, the `node_cost` and `edge_cost` predicates are extended to charge a cost of 1 per relabeling.

Listing 1.7. Graph edit distance with relabeling (modifies Listing 1.6)

```
1  {h(X,Y) : n2(Y,_)} <= 1 :- n1(X,_).
2  {h(X,Y) : n1(X,_)} <= 1 :- n2(Y,_).
3  {h(X,Y) : e2(Y,S2,T2,_), h(S1,S2), h(T1,T2)} <= 1 :- e1(X,S1,T1,_).
4  {h(X,Y) : e1(X,S1,T1,_), h(S1,S2), h(T1,T2)} <= 1 :- e2(Y,S2,T2,_).
5  ...
6  relabel_node(X,L2) :- n1(X,L1),h(X,Y), n2(Y,L2), L1 <> L2.
7  relabel_edge(X,L2) :- e1(X,_,_,L1),h(X,Y),e2(Y,_,_,L2), L1 <> L2.
8  ...
9  node_cost(X,1) :- relabel_node(X,_).
10 edge_cost(X,1) :- relabel_edge(X,_).
```

Ad Hoc Constraints. The use of ASP opens up many other possibilities for controlling or constraining the various isomorphism or edit distance problems. One example which we found useful in previous work [8] was to modify the definitions of isomorphism or subgraph isomorphism to treat properties as soft constraints and minimize the number of mismatched properties.

Another potentially interesting class of constraints is to allow “access control” constraints on the possible edit scripts, for example specifying that certain nodes or edges in one graph cannot not be modified and so must be matched with equivalent constructs in the other graph. This is similar to the approximate constrained subgraph matching problem [22].

6 Evaluation

Graph matching and edit distance are widely studied problems and a thorough comparison of our approach with state-of-the-art algorithms is beyond the scope of this paper. However, we do not claim that our approach is faster, only that it is easy to implement and modify, rendering it suitable for rapid prototyping situations. Nevertheless, in this section we summarize a preliminary evaluation that supports a claim that our approach is fast enough to be useful for rapid prototyping. Our experiments were run on an 2.6 GHz Intel Core i7 MacBook Pro machine with 8 GB RAM and using Clingo v5.2.0.

First, we consider the various problems on synthetic graphs, such as k-cycles and k-chains (linear sequences of k edges), with only one possible node and edge label and no properties. These problems are not representative of typical real problems, but illustrate some general trends. We considered each of the problems: (HOM), (ISO) $G_1 \cong G_2$, (SUB) $S_n \lesssim C_n$, and (GED) $GED(G_1, G_2)$. We first considered comparisons where G_1 and G_2 are k-cycles or k-chains, for $k \in \{10, 20, \ldots, 100\}$. We found the running times for each of these problems

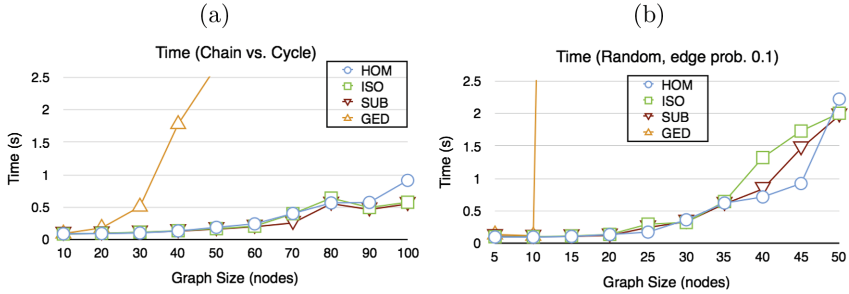

Fig. 3. Synthetic results: (a) chains and cycles (b) randomly generated graphs

to be relatively stable independent of whether the comparison was between two k-chains, a k-chain with a k-cycle, or two k-cycles, so we have averaged across all four scenarios. We also considered randomly generated graphs with k nodes and each edge generated with probability 0.1, with $k \in \{5, 10, \dots, 50\}$. We attempted each problem with a running time limit of 30 seconds; the results are shown in Fig. 3 results. Unsurprisingly, the HOM instances are solved fastest, and GED slowest.

Second, we consider some real graphs from the Mutagenesis dataset (MUTA), a standard dataset used for evaluating graph edit distance algorithms [15], for example in a recent graph edit distance competition (GEDC) [1]. In the contest, eight algorithms were run on different problems for up to 30 s, and compared in terms of time, accuracy (for approximate algorithms), and success rate (for exact algorithms). We modified the GED specification to allow node and edge relabeling and use the same weight function as in the second (and more challenging) configuration used in the contest, for which even the best algorithm (called F2) was not able to deal with graphs of size larger than 30. We consider three datasets MUTA-10, MUTA-20 and MUTA-30 each consisting of ten chemical structure graphs of size 10, 20 or 30 respectively. We also consider a dataset MUTA-MIXED which consists of ten graphs of varying sizes. We considered all unordered pairs of the graphs in each subset and attempted to find the GED with a timeout of 30 s. Table 2 shows the results compared with the four exact algorithms reported in [1]. The first two algorithms, F2 and F24threads, are implementations of a binary linear programming encoding of graph edit distance [17], the first being the plain single-threaded algorithm, and the second running with four threads. The other two, DF and PDFS, are sequential and parallel implementations of a depth-first, branch-and-bound algorithm [2,3].

Table 2 illustrates that our approach is competitive with DF and slightly worse than PDFS, but does not match the performance of the two F2 algorithms. These results should be taken with a grain of salt, since we have not replicated the GEDC results on our (slightly faster) hardware. Memory did not appear to be a bottleneck for our approach.

We have implemented and used variations of the isomorphism and subgraph isomorphism specifications for property graphs in a provenance graph analysis system called ProvMark [8]. In this earlier work, we found that for graphs of

Table 2. Success rate (optimal solution found in under 30 s) on Mutagenesis dataset

	MUTA-10	MUTA-20	MUTA-30	MUTA-MIXED
F24threads [1,17]†	100%	98%	23%	44%
F2 [1,17]†	100%	94%	15%	41%
PDFS [1,3]†	100%	26%	11%	10%
Our approach	100%	26%	10%	4%
DF [1,2]†	100%	14%	10%	10%

†Experiments from [1] run on a 4-core 2.0 GHz AMD Opteron 8350 with 16 GB RAM.

Table 3. Performance improvement vs. ProvMark [8]

Experiment	Size	Old time (s)	New time (s)	Speedup
creat-bg-gen	1006	0.060	0.034	1.9×
creat-fg-gen	1060	0.070	0.037	1.9×
creat-comp	1033	0.053	0.026	2.1×
execve-bg-gen	1006	0.061	0.036	1.7×
execve-fg-gen	1340	0.114	0.051	2.2×
execve-comp	1173	0.083	0.042	1.9×

up to around 100 nodes and edges, and a few hundred properties, these problems are usually solvable within a few seconds. However, these problems may not be representative of other scenarios.

The specifications we used to define approximate subgraph isomorphism problems in ProvMark are similar to those presented here, but we subsequently experimented with several different approaches with different performance. Here, we compare the performance of ProvMark on subgraph isomorphism problems over two representative example graphs considered in our previous experiments: the graph generalization and comparison problems resulting from benchmarking the `creat` and `execve` system calls using the CamFlow provenance recording system [20]. See [8] for further details and the Clingo code of the previous approaches.

Table 3 shows the running time of the old version and new version of approximate subgraph isomorphism. The code for both specifications is in Appendix C of the extended version [7]. The problem sizes (that is, the number of nodes, edges, and properties of the two graphs) is shown under "Size". The "Old Time" column corresponds to the time obtained using the old approach and "New Time" shows the time obtained using the code in Listing 1.5 modified to allow approximate property matching. The "Speedup" column shows the ratio between the old and new time. In most cases, the speedup is around a factor of two. As future work, we plan to use graph edit distance with the results of the ProvMark system, for example for clustering or regression testing across runs.

7 Related Work

The lower bound of the complexity of graph isomorphism is a well-known open problem [4], but subgraph isomorphism and graph edit distance are NP-complete [12]. A number of practical algorithms for graph isomorphism have been studied, however, including NAUTY [18], which has also been integrated with Prolog [10]. However, most such algorithms consider graphs with vertex labels but not edge labels or properties, so are not directly applicable to property graph isomorphism. Subgraph isomorphism has been studied extensively over the past years, one survey [19] summarizes the state-of-art algorithms for solving partial or simplified version of the problem. Subgraph isomorphism is also studied for graph databases, where the query subgraph is usually small but the other graph may be very large. Lee et al. [16] evaluated five such algorithms on query graphs of up to 24 edges and databases of up to tens of thousands of nodes and edges. Approximate subgraph matching with constraints has also been studied, particularly in biomedical settings [22], and it would be interesting to investigate whether our approach is competitive with their CSP-based algorithm. Graph edit distance has also been studied extensively [11], with much attention on approximate algorithms that can provide results quickly [21].

While several approaches to graph matching and edit distance have been based on expressing these problems as constraint satisfaction problems, satisfiability, or linear programming problems, to the best of our knowledge there is no previous work based on answer set programming. Moreover, our approach easily accommodates richer graph structure such as hard or soft label constraints, properties, and multiple edges between pairs of nodes, whereas the algorithms we have seen generally consider ordinary graphs (without properties and with at most one edge between two nodes).

8 Conclusions

The graph edit distance problem is a widely studied problem that has many applications. Exact solutions to it, and to related problems such as graph isomorphism and subgraph isomorphism, are challenging to compute efficiently due to their NP-completeness or unresolved complexity (in the case of graph isomorphism). There are a number of proposed algorithms in the literature, with one of the most effective based on a reduction to binary linear programming [17]. In this paper, we investigated an alternative approach using answer set programming (ASP), specifically the Clingo solver. This approach may not be competitive with the best known techniques in terms of performance, but has the potential advantage that it is straightforward to modify the problem specification to accommodate different kinds of graphs, cost metrics or other variations, or to accommodate ad hoc constraints that can also be expressed using ASP. Our approach has already proved useful for a real application [8], and our experimental evaluation suggests that it is also competitive with two out of four exact algorithms from a graph edit distance competition.

Our work may be valuable to others interested in rapid prototyping of graph matching or edit distance problems using declarative programming. Additional work could be done to facilitate this, for example using Clingo's Python wrapper library. Graph matching and edit distance problems may also be an interesting class of challenge problems for developers of ASP solvers.

Acknowledgments. Effort sponsored by the Air Force Office of Scientific Research, Air Force Material Command, USAF, under grant number FA8655-13-1-3006. The U.S. Government and University of Edinburgh are authorised to reproduce and distribute reprints for their purposes notwithstanding any copyright notation thereon. Cheney was also supported by ERC Consolidator Grant Skye (grant number 682315). This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under contract FA8650-15-C-7557.

References

1. Abu-Aisheh, Z., et al.: Graph edit distance contest: results and future challenges. Pattern Recogn. Lett. **100**, 96–103 (2017)
2. Abu-Aisheh, Z., Raveaux, R., Ramel, J.-Y., Martineau, P.: An exact graph edit distance algorithm for solving pattern recognition problems. In: Proceedings of the International Conference on Pattern Recognition Applications and Methods (ICPRAM 2015), pp. 271–278 (2015)
3. Abu-Aisheh, Z., Raveaux, R., Ramel, J.-Y., Martineau, P.: A parallel graph edit distance algorithm. Expert Syst. Appl. **94**, 41–57 (2018)
4. Arvind, V., Torán, J.: Isomorphism testing: perspectives and open problems. Bull. EATCS **86**, 66–84 (2005)
5. Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z.: DBpedia: a nucleus for a web of open data. In: Aberer, K., et al. (eds.) ASWC/ISWC -2007. LNCS, vol. 4825, pp. 722–735. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-76298-0_52
6. Bunke, H.: On a relation between graph edit distance and maximum common subgraph. Pattern Recogn. Lett. **18**(8), 689–694 (1997)
7. Chan, S.C., Cheney, J.: Flexible graph matching and graph edit distance using answer set programming (extended version). CoRR, abs/1911.11584 (2019)
8. Chan, S.C., et al.: ProvMark: a provenance expressiveness benchmarking system. In: Proceedings of the 20th International Middleware Conference (Middlware 2019), pp. 268–279. ACM (2019)
9. Chen, X., Huo, H., Huan, J., Vitter, J.S.: An efficient algorithm for graph edit distance computation. Knowl.-Based Syst. **163**, 762–775 (2019)
10. Frank, M., Codish, M.: Logic programming with graph automorphism: integrating nauty with prolog (tool description). TPLP **16**(5–6), 688–702 (2016)
11. Gao, X., Xiao, B., Tao, D., Li, X.: A survey of graph edit distance. Pattern Anal. Appl. **13**(1), 113–129 (2010)
12. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman, New York (1979)
13. Gebser, M., et al.: The potsdam answer set solving collection 5.0. KI-Künstliche Intelligenz **32**(2–3), 181–182 (2018)
14. Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., Schneider, M.T.: Potassco: the Potsdam answer set solving collection. AI Commun. **24**(2), 107–124 (2011)

15. Kazius, J., McGuire, R., Bursi, R.: Derivation and validation of toxicophores for mutagenicity prediction. J. Med. Chem. **48**(1), 312–320 (2005)
16. Lee, J., Han, W.-S., Kasperovics, R., Lee, J.-H.: An in-depth comparison of subgraph isomorphism algorithms in graph databases. PVLDB **6**(2), 133–144 (2012)
17. 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)
18. McKay, B.D.: Practical graph isomorphism. Congressus Numerantium **30**, 45–87 (1981)
19. McKay, B.D., Piperno, A.: Practical graph isomorphism, II. J. Symb. Comput. **60**, 94–112 (2014)
20. Pasquier, T., et al.: Practical whole-system provenance capture. In: Proceedings of the 2017 Symposium on Cloud Computing (SoCC 2017), pp. 405–418 (2017)
21. Riesen, K.: Structural Pattern Recognition with Graph Edit Distance - Approximation Algorithms and Applications. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-27252-8
22. Zampelli, S., Deville, Y., Dupont, P.: Approximate constrained subgraph matching. In: Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP 2005), pp. 832–836 (2005)