Benchmarking Whole Knowledge Graph Embedding Techniques

Pieter Bonte, Sander Vanden Hautte, Filip De Turck, Sofie Van Hoecke, Femke Ongenae

IDLab, Ghent University - imec
Technologiepark-Zwijnaarde 126
B-9052, Ghent, Belgium
Pieter.Bonte@ugent.be

Knowledge graphs (KGs) are gaining popularity and are being widely used in a plethora of applications. They owe their popularity to the fact that KGs are an ideal form to integrate and retrieve data originating from various sources. Using KGs as input for Machine Learning (ML) tasks allows to perform predictions on these popular graph structures. However, KGs cannot directly be used as ML input in their graph representation, they first require to be transformed to a vector space representation through an embedding technique. As ML techniques are data-driven, they can generalize over unseen input data that deviates to some extent from the data they were trained upon. To fully exploit the generalization capabilities of ML algorithms when using embedded KGs as input, small changes in the KGs should also result in small changes in the embedding space. Various embedding techniques for graphs in general exist, however, they have not been tailored towards embedding whole KGs, while KGs can be considered a special kind of graph that adheres to a certain KG schema.

This paper evaluates if these existing embedding techniques that embed the whole graphs can represent the similarity between KGs in their embedding space, allowing ML algorithms to generalize over their input. We compare the similarities between KGs in terms of changes in size, entity labels, and KG schema. We found that most techniques were able to represent the similarities in terms of size and entity labels in their embedding space, however, none of the techniques were able to capture the similarities in KG schema.

Keywords: Knowledge Graphs, Embeddings, Graph Comparison, Benchmark

1. Introduction

Knowledge Graphs (KGs) are increasing in popularity. They are the ideal form to structure and integrate information originating from various sources. Popular KGs include YAGO [1], DBpedia [2], Freebase [3] or the Google Knowledge Graph [4]. Machine Learning (ML) techniques can be employed to further fine-tune and augment these KGs. Through link prediction, i.e. predicting certain connections between entities in the KG, node classification, i.e. predicting the label of a certain node [5], or (sub)graph classification, i.e. assigning a label to the whole graph or a subgraph [6].

However, there is a mismatch between the graph structure of KGs and the input required for ML algorithms. Embedding techniques learn to convert the graph structure to a vector representation, allowing them to be used as ML input. Various KGs embedding techniques exist, however, focusing on embedding only a specific

part of the KG, i.e. the nodes or the edges [7]. Embedding whole KGs has, until now, received little attention and is the focus of this paper. Embedding techniques for whole graphs that are not KGs exist, however, it is unclear how they translate to KGs. These graphs differ from KGs in the sense that their structure is not as complex as a KG, e.g. their edges have no labels, their edges are undirected or they do not adhere to a certain schema, as a KG does.

To be useful for ML tasks, similar KGs should result in embeddings that do not deviate too much. This allows ML algorithms to generalize over unseen data. It is thus important that closely related KGs are represented close together in embedding space, compared to KGs that deviate a lot. To evaluate this property, we focus on KG comparison through graph embedding techniques, as similar KGs would result in closely related embeddings. We note that (Knowledge) Graph comparison by itself has many use cases, such as computer security where control flow graphs need to be compared to detect vulnerabilities [8], comparison of chemical compounds [9] or comparing communities in social networks [10].

This paper evaluates if graph embedding techniques, that were designed for graphs in general, can maintain the similarities between the more complex KGs in their embedding space. We evaluate these similarities in embedding space in three dimensions: the size of the KGs, the labels of the entities, and the used schema.

The paper is structured as followed: In Section 2, we first detail the background needed to understand the remainder of the paper. Section 3 details the used graph embedding techniques, while Section 4 will compare the differences between these techniques in more detail over various comparison tasks. In Section 5 we detail a thorough discussion on the differences between the graph embedding techniques for their use on KGs. We summarize the findings in Section 6 and provide suggestions for further research.

2. Preliminaries

This section details all needed preliminary knowledge for the remainder of the paper.

2.1. *Graph types*

There exist three types of graphs, each with different properties:

Definition 1. A **Graph** $\mathcal{G} = (V, E)$, with V the set of nodes and E the set of edges. $\mathcal{G}$ is associated with a node type mapping function $f_v : V \to \mathcal{T}^v$ and an edge type mapping function $f_e : E \to \mathcal{T}^e$. The function l is a labeling function that maps each node or edge onto its corresponding label or type.

$\mathcal{T}^v$ and $\mathcal{T}^e$ respectively denote the set of node types and edge types. The different graph types are defined as:

- A *Homogeneous graph* $\mathcal{G} = (V, E)$ is a graph in which $|\mathcal{T}^v| = |\mathcal{T}^e| = 1$. This means that all nodes and edges belong to the same type. We also

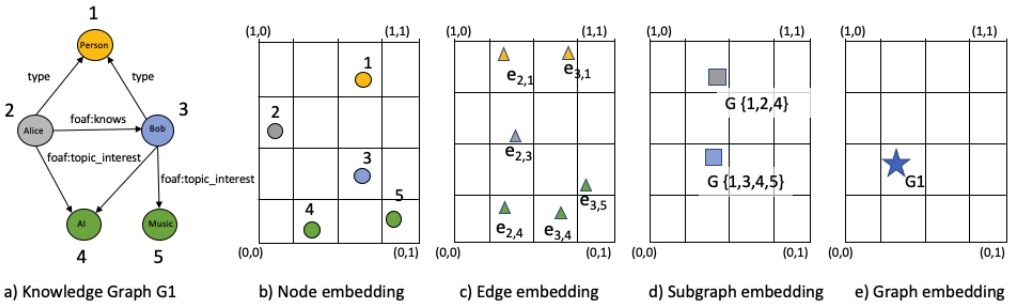


Fig. 1: Different embedding techniques.

refer to the homogeneous graph as the *unlabeled graph*. The edges of the homogeneous graph can be both directed and undirected.

- A *Heterogeneous graph* $\mathcal{G} = (V, E)$ is a graph in which $|\mathcal{T}^v| \geq 1$ and $|\mathcal{T}^e| \geq 1$. Each node and edge can thus have different a type. We refer to the heterogeneous graph as *labeled graph* when $|\mathcal{T}^v| \geq 1$ and $|\mathcal{T}^e| = 1$ and f_v is a surjection. This means that nodes can have different types, however each node has only one type, also called the label of the node. Furthermore, labeled graphs do not support edge labels. The edges of the heterogeneous graph can be both directed and undirected.
- A *Knowledge graph* $\mathcal{G} = (V, E)$ is a special kind of heterogeneous graph composed of triples. Each triple consists of a (subject, property, object) relation, where subjects and objects represent nodes and properties represent edges. KGs typically adhere to a certain schema, meaning that the labels of their nodes and edges are structured in some kind of hierarchy. The edges of a KG are always directed.

In Figure 1 a), an example KG describes two persons, i.e. *Alice* and *Bob*, using the Friend of a Friend (foaf)[a] schema:
`(Alice, type, Person)`,
`(Bob, type, Person)`,
The KG also details that *Alice knows Bob*:
`(Alice, foaf:knows, Bob)`,
And that *Alice* is interested in Artificial Intelligence (AI) and that *Bob* is interested in both AI and music.
`(Alice, foaf:topic_interest, AI)`,
`(Bob, foaf:topic_interest, AI)`,
`(Bob, foaf:topic_interest, Music)`
Combining these triples results in a graph, as visualized in Figure 1 a).

[a]http://xmlns.com/foaf/spec/

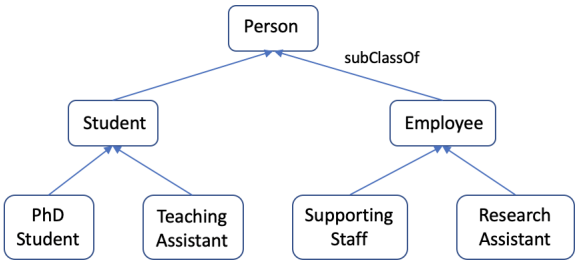


Fig. 2: Example of KG schema detailing a Person concept hierarchy.

2.2. *KG schema*

As explained above, the nodes and edges in a KG can adhere to a certain schema. Nodes can have a certain *type*, e.g. Alice is of the type 'Person', while edges can adhere to certain *properties*, e.g. Alice 'knows' Bob. The schema fixes the semantics of the graph. The schema itself can be composed in a hierarchy, e.g. Figure 2 shows a concept hierarchy of different kinds of 'Persons'. This means that entities of the type 'Student' are also of the type 'Person'. Properties can also be defined in terms of hierarchies and much more expressive relations between types and hierarchies can be defined in ontology languages such as the Web Ontology Language (OWL) [18].

2.3. *Embedding types*

There exist four different types of graph embeddings, each type is depicted in Figure 1 b) - e):

- *Node embeddings* represent each node as a vector in a low dimensional space.
- *Edge embeddings* on the other hand transform edges in low dimensional vector space.
- *Subgraph embeddings* represent a selection of nodes and edges as vectors.
- *Graph embeddings* transform whole graphs into a single vector.

Nodes, edges, subgraphs, or graphs that are "closely related" have similar vector representation in each of their embedding space. This has been shown in general for homogeneous and labeled graphs [11], while for KGs only node and edge embeddings have so far been proved to maintain the relatedness-property [12, 13, 14], where related node/edges have similar representations in vector space.

In this paper, we focus specifically on whole KG embeddings.

2.4. *From Knowledge Graphs to Labeled Graphs*

KGs can be converted to labeled graphs through a conversion technique proposed by de Vries et al. [15]. This allows us to evaluate if graph embedding techniques

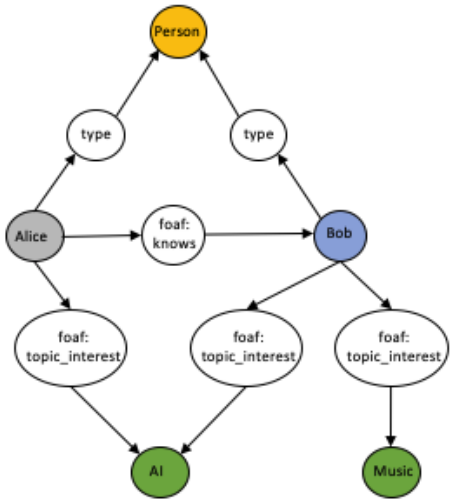


Fig. 3: Representation of the RDF graph without edge labels.

that were designed specifically for labeled graphs, i.e. graphs that only allow node labels, are able to maintain the similarity between KGs in vector space. de Vries et al. [15] propose a technique to represent the KG data without edge labels. For each triple, two edges are created. A label function l assigns a label to each entity in a triple. $l(s) = s$ and $l(o) = o$ for each subject/object in a triple (s, p, o) and the label $l((s, p, o)) = p$ for each property. Figure 3 shows this representation of the graph introduced in Figure 1. However, the graph doubles in size due to the conversion process, as can be seen in Figure 3.

2.5. *Path extraction*

As we will see in Section 3, some graph embedding techniques extract paths from the graphs as preprocessing step. A path (or walk) is a sequence of nodes that can be extracted from the graph by traversing the (directed) edges. We can notate a path of nodes of length n as follows:

$$v_0 \rightarrow v_1 \rightarrow ... \rightarrow v_{n-1}$$

Using the labeling function l, a sequence of labels can be created and obtained by traversing the graph:

$$l(v_0) \rightarrow l(v_1) \rightarrow ... \rightarrow l(v_{n-1})$$

Various path extraction algorithms exist, we detail the Weisfeiler-Lehman (WL) path extraction technique [20] as it is used by many other techniques in Section 3.

WL is a relabeling process that does h iterations of: 1) looking up the neighbors of a node, 2) sorting the labels of the neighbors, 3) concatenating the label of the

original node with sorted labels of the neighbors and 4) assigning a new label to the concatenation and assign the new label to the original node. Thus next to extracting a path, WL also summarizes information of the node's neighborhood.

2.6. *Graph Edit Distance*

Graph Edit Distance (GED) [16] calculates the edit distances between graphs directly, allowing the graphs to be compared without an embedding step. As we will see in Section 3, some supervised techniques use the GED as ground truth for their learning task. The GED between two graphs G_1 and G_2 is the minimum number of edits to transform G_1 into G_2. An edit operation consists of inserting or deleting a node/edge or relabeling a node. Two identical graphs have a GED of 0. However, the exact GED between graphs with more than 16 nodes cannot be reliably computed within a reasonable amount of time [17]. This renders GED techniques unusable for the comparison of real-world graphs as they greatly exceed the 16 nodes limit.

3. Related work

Many graph embedding techniques exist, operating on different types of graphs and producing various types of embeddings. To limit the scope, we focus specifically on embedding techniques which take as input directed heterogeneous graphs and output whole graph embeddings. As KGs cannot be transformed into homogeneous graphs without losing information, we do not take these techniques into account. In Section 2.4, we explained how KGs can be converted to labeled graphs. Note that embedding techniques for whole KGs do not exist, therefore we evaluate in this paper if the conversion of existing embedding techniques for heterogeneous graphs allows to maintain the similarities between KGs in vector space. Three graph embedding categories can be distinguished in the literature: graph kernel approaches, unsupervised approaches, and supervised deep learning approaches. Table 1 summarizes the various techniques. We note that none of these techniques were designed for KGs.

3.1. *Graph kernel approaches*

Graph kernels utilize a similarity function to capture the semantics of the graph structures [19]. This is often done through aggregating the similarity measures between graph fragments, e.g. through path extraction.

Weisfeiler-Lehman Graph Kernels [20] exploit the WL relabeling process to optimally describe the neighborhood of each node in the graph. The kernel will generate, for each graph, a vector detailing both the counts of each of the original and new node labels. As the vector contains the counts of the labels, and thus the neighborhoods, in each of the graphs, it represents the similarity between graphs. However, the size of the vector cannot be fixed, larger graphs will result in larger vectors, making the technique not ideal for ML tasks.

Deep Graph Kernels [10] extracts substructures using techniques such as WL, but uses the neural document embedding model *Word2vec* [21] to learn similarities of these substructures among graphs. Once these substructures have been computed, they use the typical graph kernel approaches to calculate the similarity between graphs. The kernel is computed as:

$$\mathcal{K}(\mathcal{G}, \mathcal{G}') = \phi(\mathcal{G})^T \mathcal{M} \phi(\mathcal{G}') \tag{1}$$

With $\phi(\mathcal{G})$ the vector representation of G and $\mathcal{M}$ a matrix representing the similarity between the substructures of $\mathcal{G}$ and $\mathcal{G}'$. Instead of computing the similarity matrix through edit-distances, Deep Graph Kernels computes this similarity matrix by learning latent representations of the substructures. This means that $\mathcal{M}$ is computed through *Word2Vec*. The extracted substructures in each iteration of WL are treated as co-occuring and thus seen as similar by the *Word2Vec* algorithm. The resulting matrix $\mathcal{M}$ contains the embedding of each of the substructures on its diagonal. By multiplying the vector representation of each graph with $\mathcal{M}$, a unique graph embedding can be obtained. However, the size of the embedding depends on the number of graphs that are being compared.

Graph kernels approaches are ideal for comparing graphs, however, less optimal for usage in ML tasks, as their embeddings depend on the size of the graphs or the number of graphs being used. We note that both techniques are optimized for labeled graphs.

3.2. *Unsupervised graph embedding approaches*

Graph2Vec [9] is a neural embedding framework, inspired by the neural document embedding model *Doc2Vec* [22]. Graph2Vec views an entire graph as a document and the rooted subgraphs around every node in the graph as words that compose the document. Once the rooted subgraphs have been extracted, the standard document embedding procedures can be used to embed the graph. In order to generate the subgraphs, Graph2Vec uses the WL relabeling process [20]. The result of Graph2Vec is a task agnostic graph embedding. Similar to *Doc2Vec*, the embeddings can be obtained by extracting the internal representation of the document embedding network.

Graph Embedding with Frequent Sub-Graphs (GE-FSG) [23] takes a similar approach, but instead of extracting rooted subgraphs, it looks for frequent subgraphs. This allows to better capture the relations between graphs. However, finding the frequent subgraphs is a mining process that can become very expensive. The embeddings are also obtained by extracting the internal representation of the document embedding network.

Unsupervised graph embedding approaches can produce fixed-length vectors, making them ideal for ML tasks, however, they are less optimized for graph comparison tasks. Both Graph2Vec and GE-FSG operate on labeled graphs.

3.3. *Supervised deep learning graph embedding approaches*

SimGNN is a supervised deep learning approach that predicts the GED between graphs. It aggregates the node embeddings obtained by Graph Convolutional Networks (GCNs) [24] and adds an attention mechanism to visualize the graph differences. In order to train, it requires the GED between graphs, which is expensive to obtain for larger graphs. We will explain GCNs in its simplest form. A GCN is a neural network containing multiple hidden layers that aggregate, in each hidden layer, the representation of the neighbors in the nodes.

In its simplest form[b] a GCN can be defined as:

$$H^{(l+1)} = \sigma(AH^{(l)}W^{(l)}) \tag{2}$$

With A the adjacency matrix, $W^{(l)}$ the learned weights for the l-th layer of the network and σ a non-linear activation function. It can be seen as a generalization of WL as in each hidden layer, it summarizes the information in the neighboring nodes for each node in the graph. The next layer will use the summarizations of the previous layer. The number of hidden layers in the GCN is thus related to the number of iterations in WL.

The GCN produces a summarization (or embedding) for each node in the graph. SimGNN converts the node embeddings to a whole graph embedding by employing an attention mechanism. An attention mechanism will learn which nodes should receive more weight when summarizing the different node embeddings into a whole graph embedding.

SimGNN operates on labeled graphs, however, requires knowing the GED before training, which is very expensive to compute.

Graph Matching Networks [8] takes a similar approach to predict the similarity between graphs, but provides a framework to adapt the graphs, by adding, removing and relabling nodes and edges. This eliminates the need to calculate the real GEDs. GMN uses a Graph Neural Network (GNN) instead of GCN that uses message passing to aggregate the information of each nodes local neighborhood. First, node and edge features are encoded:

$$\begin{aligned} h_i^{(0)} &= MLP_{node}(x_i), \forall i \in V \\ e_{ij} &= MLP_{edge}(x_i j), \forall (i,j) \in E \end{aligned} \tag{3}$$

With MLP a Multilayer Perceptron (MLP), i.e. a classic feedforward neural network. A propagation layer maps the nodes to a new representation in each layer, as follows:

$$\begin{aligned} m_{j \to i} &= f_{message}(h_i^{(t)}, h_j^{(t)}, e_{ij}) \\ h_i^{(t+1)} &= f_{node}(h_i^{(t)}, \sum_{j:(j,i) \in E} m_{j \to i}) \end{aligned} \tag{4}$$

[b]In the extended form, self-loops are included by adding the identity matrix to the adjacency matrix and another matrix is added to the equation to normalize the node degree.

Table 1: Summary of the investigated Graph embedding techniques.

Name	Input	Output	Category
Weisfeiler-Lehman Graph Kernels	labeled graph	Graph comparison	Kernel
Deep Graph Kernels	labeled graph	Graph comparison	Kernel
Graph2Vec	labeled graph	Graph embedding	Unsupervised
Graph Embedding with Frequent Sub-Graphs (GE-FSG)	labeled graph	Graph embedding	Unsupervised
SimGNN	labeled graph	Graph Similarity	Supervised
Graph Matching Networks	heterogeneous graph	Graph Similarity	Supervised

With $f_{message}$ an MLP and f_{node} either an MLP or a recurrent neural network. Thus each layer will use message passing to summarize the information of each node's neighborhood. In the message-passing phase, each node will aggregate the information (of the previous layer) of its neighboring nodes. It will use different weights for each edge, and can thus handle edge features as well, in comparison to GCN. GMN uses an extension of the GNN technique that also performs matching between graphs, to better capture the differences between graphs. Similar to SimGNN, in the last step, GNN aggregates the node embedding to a whole graph embedding. As the GNN used in GMN can encode edge features as well, GMN can operate on heterogeneous graphs.

3.4. *Conclusion*

No techniques exist to directly embed KGs without some sort of conversion to lesser expressive graphs, such as labeled graphs. Table 1 provides an overview and compares the techniques in terms of graph types that can be used as the input, the output, and the category the technique belongs to. The kernel techniques provide as output a graph comparison, however, as we have detailed above, embeddings can be extracted for these techniques as well. The supervised techniques calculate a graph similarity score as output, but also for these techniques, the intermediate embeddings can be extracted.

4. Graph Embedding Comparison Benchmark

This section describes how the various embedding techniques will be compared. The framework for the evaluation and comparison of the techniques is available on Github[c].

4.1. *Comparing KGs*

As KGs are complex graph structures, we will investigate which different kinds of changes in the KG the embedding techniques can detect and represent the similar-

[c]https://github.com/IBCNServices/KGEmbeddingBenchmark

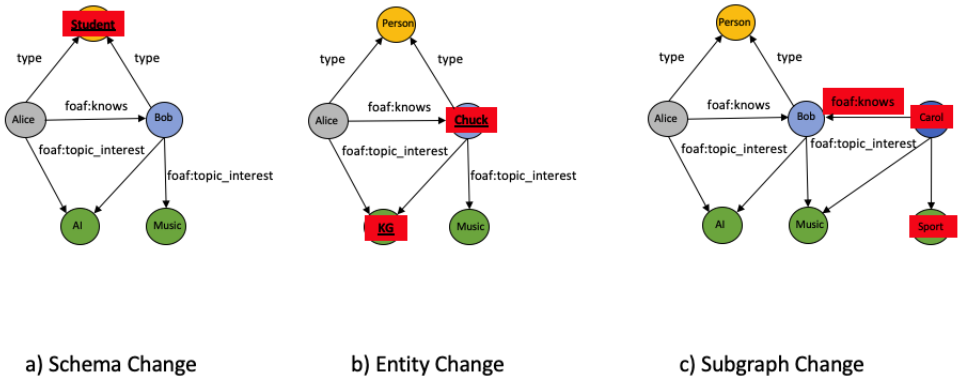


Fig. 4: Different KG changes.

ities in their vector space. The changes that can occur in a KG can be categorized as follows, as depicted in Figure 4:

- *Schema changes:* The entities and their neighbors remain the same, however, the used schema changes. This means that the classes and properties change.
- *Entity changes:* The used schema and connectivity between the entities remain the same, however, the entities labels themselves change.
- *Subgraph changes:* The KG grows or shrinks in size through adding or remove nodes or edges.

We differentiate between three different kinds of changes:

- *Addition:* An entity or property is added to the graph.
- *Removal:* An entity or property is removed from the graph.
- *Substitution:* An entity or property is replaced in the graph. This is a combination of *removal* and *addition*.

4.2. *Datasets*

We have selected two open KGs datasets, which are commonly used in the literature for the evaluation of entity embeddings, however, we are the first to exploit these datasets for whole KG embedding evaluation:

- MUTAG[d]: describes information about complex molecules that are potentially carcinogenic. Each description of a molecule is a distinct graph.

[d]http://data.dws.informatik.uni-mannheim.de/rmlod/LOD_ML_Datasets/data/datasets/RDF_Datasets/MUTAG/

Table 2: Summary of the data sets, detailing the number of graphs in the data set, the average (AVG) triples of each graph, the average number of common triples between graphs and the percentage of the number of common triples.

Dataset Name	**#graphs**	**AVG #triples**	**AVG intersection size**	**AVG % intersection**
MUTAG	340	266.9	30	11.2%
BGS	11697	179.6	76.7	42.7%

- AIFB[e]: describes the AIFB research institute in terms of its staff, research group, and publications. Each graph describes a research group within the institute.

Table 2 summerizes the characteristics of the datasets.

4.3. *Evaluation Set-up*

The evaluation focuses on detecting various possible changes in KGs, as explained in Section 4.1. We investigate how the different techniques handle these changes. We tailored the various evaluations such that the x-axis indicates an increase in distances when comparing the KGs. Table 3 and Table 4 summarize the increase rates, respectively for the MUTAG and BGS datasets, for the different techniques on the various evaluations explained below. The *increase rate* is defined as the percentage of increase in sequential values. A linearly increasing line would thus result in an increase rate of 100%. Note that exponential or logarithmic increases in distances also result in an increase rate of 100%. We included this flexibility to allow different embedding techniques the flexibility to react differently on various graph changes.

4.3.1. *Gradual Changes Set-up*

In this evaluation, we select two random graphs (G_1 and G_2) from the dataset and gradually convert G_1 into G_2. This is done by removing the triples in G_1 that are in G_1 but not in G_2 and adding the triples that are in G_2 but not in G_1. We take special measures when converting the graphs, that the graphs stay connected and a removal of a triple in G_1 does not results in two disconnected subgraphs. This is done by checking the connectivity after the removal of each triple. If the graph is not connected anymore, we re-add the removed triple and select another triple to be removed. Algorithm 1 gives an overview of this process in pseudocode[f]. As the

[e]`http://data.dws.informatik.uni-mannheim.de/rmlod/LOD_ML_Datasets/data/datasets/RDF_Datasets/AIFB/`

[f]Note that the variable counter makes sure that the algorithm does not get stuck in an endless loop when the graphs cannot be converted into each other without breaking the connectivity of

graphs are gradually changing, we expect to see the distances between the graphs in vector space becoming gradually larger as well.

```
Data: Graph G1 and G2
Result: List gradualList containing conversion of G1 into G2
gradualList:=[];
g1Deletes = G1\G2 //Triples in G1 that are not in G2 can be removed;
g1Adds = G2\G1 //Triples in G2 that are not in G1 should be added;
while len(g1Deletes) >0 or len(g1Adds)>0 do
    counter:=0;
    //Find triple to remove that does not break the connectivity of the
     graph;
    while len(g1Deletes)>0 and counter <100 do
        rmTriple = sample(g1Deletes);
        G1.remove(rmTriple);
        if G1.isConnected() then
            g1Deletes.remove(rmTriple)
        else
            G1.add(rmTriple)
        end
        counter+=1
    end
    counter:=0;
    //Find triple to add that does not break the connectivity of the graph;
    while len(g1Adds)>0 and counter < 100 do
        addTriple = sample(g1Adds);
        G1.add(addTriple);
        if G1.isConnected() then
            g1Adds.remove(addTriple)
        else
            G1.remove(addTriple)
        end
        counter+=1
    end
    gradualList.add(G1)
end
```

Algorithm 1: Gradual conversion of Graph G1 into G2.

the graph.

4.3.2. *Schema Changes Set-up*

As KGs typically contain some kind of schema, we will evaluate if the embedding techniques can take the schema information into account. Specifically, we will evaluate if the techniques can take the concept hierarchies of these schemas into account. First, we will gradually substitute the class of each individual with its superclass and compare this with a scenario where we substitute with a random class. As classes are very much related with their sub- or superclasses through these schema definitions (see Section 2.2), we expect to see smaller differences in embedding space when substituting a class with its superclass compared to substituting with a random class. If no superclass is available, we artificially extend the schema to enforce that each used class has a superclass.

4.3.3. *Entity Changes Set-up*

As Knowledge Graphs are defined by their schema, we evaluate how dependent the different embedding techniques are on the exact name of the entities. Note that in ontology languages such as OWL the Unique Name Assumption (UNA) does not hold, which means that entities with different names can refer to the same instance. We evaluate what happens when we gradually substitute the entity names in a Knowledge Graph. As the graphs are gradually changing in terms of entities, we expect to see the distances between the graphs in vector space becoming gradually larger as well.

4.3.4. *Time comparison Set-up*

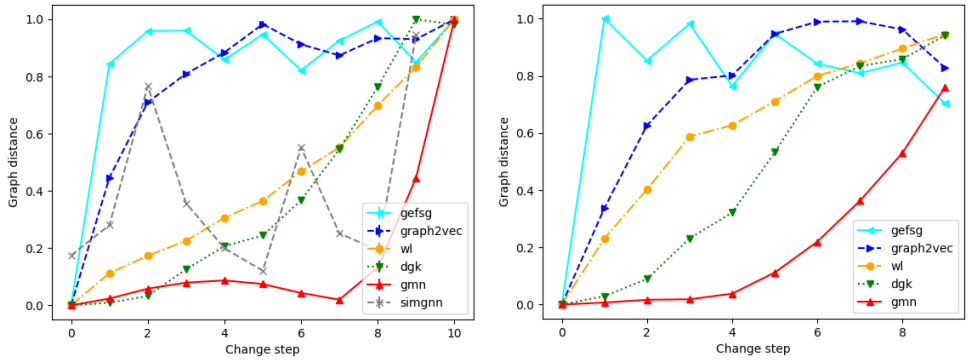
We evaluate the execution time for the different embedding techniques. We make a distinction between comparing 10 and 100 graphs. We made a split-up between the time to generate the embeddings and the time needed to compare the embeddings and calculate the distance scores.

4.4. *Evaluation Results*

We now detail the results of each evaluation. In some of the evaluation results, SimGNN or GE-FSG are not depicted, this is due to runtime timeouts or internal errors as these techniques showed to be less stable.

4.4.1. *Gradual Changes Results*

Figure 5 depicts the distances in embedding space between the original graph and the gradual changing graph for each technique for the MUTAG and BGS dataset respectively. We see that except for SimGNN, all techniques show an increase in distance. Graph2Vec and GE-FSG show a more logarithmic increase, DGK, and GMN show a more exponential increase while WL shows a linear increase. However,



(a) MUTAG dataset (b) BGS dataset

Fig. 5: Influence on the graph distances of converting one graph into another by adding and deleting triples.

the fact that Graph2Vec and GE-FSG have a decreasing factor as well, results in a lower increase rate, as can be seen in Table 3 and 4.

Figure 6 reports the distances when only adding triples, while Figure 7 visualizes the distances when only removing triples. We make the distinctions because when substituting (adding and removing) the number of triples stays roughly the same, while in the case of adding and removing the graphs change in size. We can see that GMN performs a lot better when graphs have different sizes.

4.4.2. *Schema Changes Results*

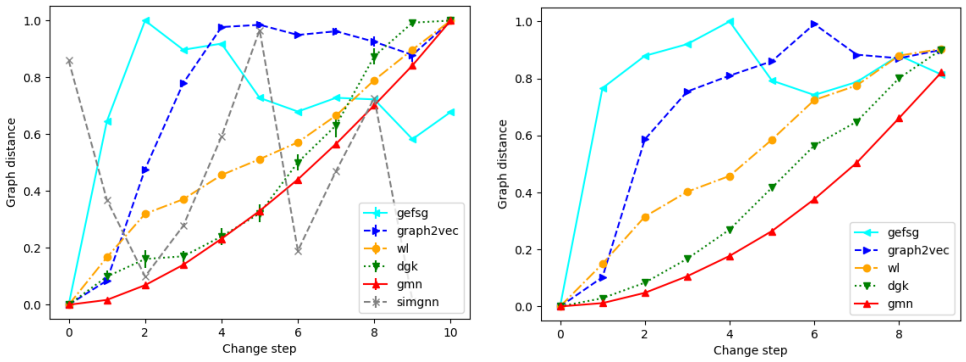
Figure 8 shows the influence of changing classes according to the schema hierarchy for the MUTAG and BGS dataset. Figure 9 shows the influence of substituting random classes for the MUTAG and BGS dataset. We see that the schema changes and random changes basically show very similar behavior. This indicates that the embedding techniques are not able to identify schema changes.

4.4.3. *Entity Changes Results*

Figure 10 shows the influence of substituting the entity names in the KGs, respectively for the MUTAG and BGS dataset. We see that all embedding techniques are able to detect entity changes relatively well.

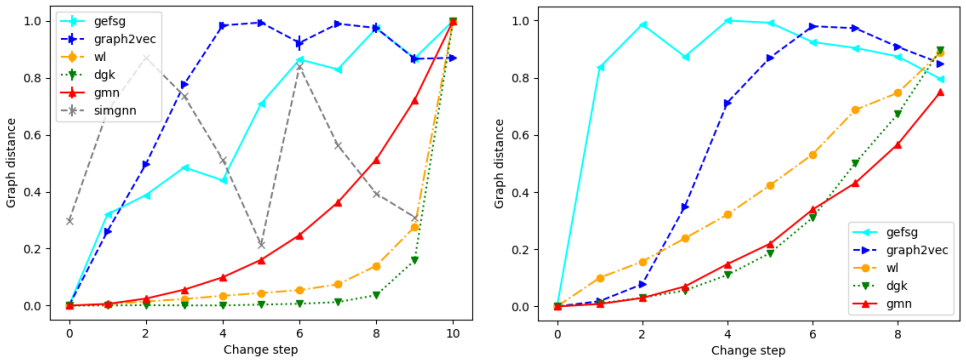
4.4.4. *Time comparison Results*

Table 5 and 6 show a comparison between the execution time (in seconds) for the different embedding techniques for the MUTAG and BGS dataset. The time to generate the embeddings is indicated as ‘Embedding’ and the time needed to compare the embeddings and calculate the distances scores is indicated as ‘Comparing’.



(a) MUTAG dataset (b) BGS dataset

Fig. 6: Influence of converting one graph into another and only adding additional triples on the graph distances.

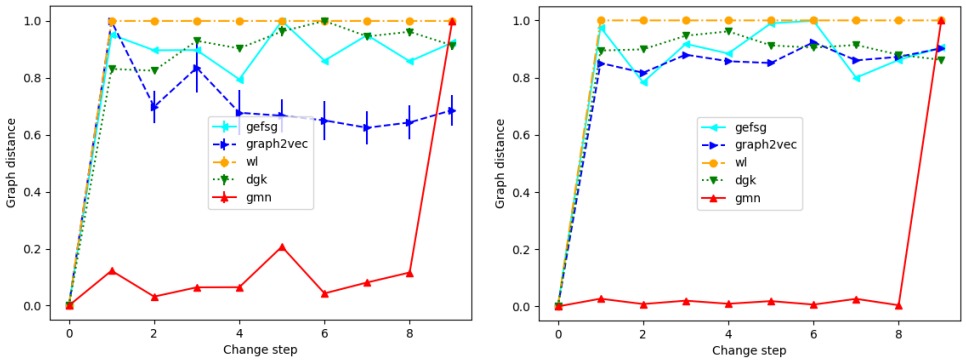


(a) MUTAG dataset (b) BGS dataset

Fig. 7: Influence of converting one graph into another and only deleting unnecessary triples on the graph distances.

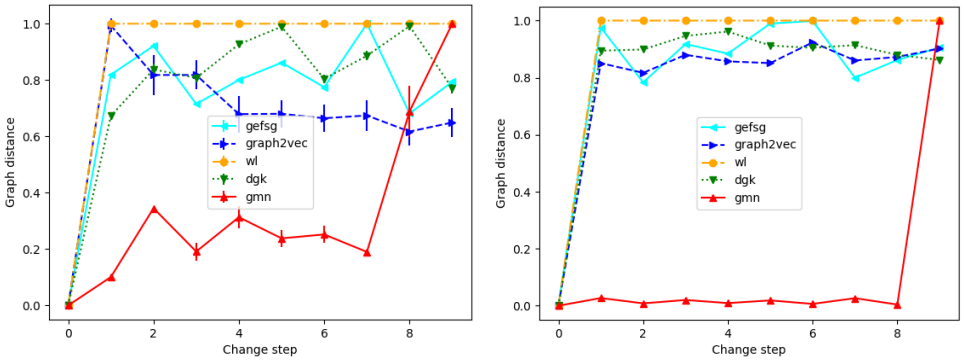
GE-FSG is rather slow for 10 graphs but scales well to larger amounts of graphs. GE-FSG depends mainly on the time to extract frequent subgraphs. However, the number of extracted graphs depends on the support threshold. Furthermore, the more frequent edges, the worse the subgraph algorithm performs. The complexity of gSpan, the frequent subgraph extraction algorithm used in GE-FSG is $O(2^{(2f)})$ with f the number of frequent edges in the graph.

Graph2Vec is built on Doc2Vec and WL. Doc2vec typically scales linearly in the number of documents. The complexity of the Weisfeiler-Lehman path extraction is O(hm) with m the number of edges and h the number of iterations. Graph2Vec thus results in a complexity of $O(Nhm)$ with N the number of graphs. DGK roughly



(a) MUTAG dataset (b) BGS dataset

Fig. 8: Influence of gradually changing the schema of the graph.



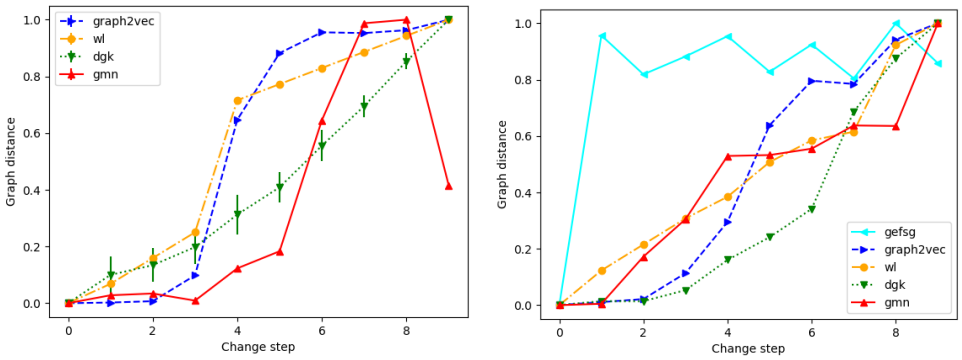
(a) MUTAG dataset (b) BGS dataset

Fig. 9: Influence of randomly changing the schema of the graph.

shares the same complexity as Graph2Vec but utilizes word2vec instead of Doc2vec.

We see that WL is very fast at comparing small amounts of graphs, however, scales badly when the number of graphs increases. This is because the complexity of the WL kernel, which includes the construction of the occurrence vectors (which are not required for Graph2Vec and DGK) is $O(Nhm + N^2hn)$, with N the number of graphs, n the number of nodes and h the number of WL iterations [20]. Thus, quadratic in the number of graphs.

GMN and SimGNN scale linear in the number of graphs, however, have a larger complexity to train in the comparison networks in general. We see that the comparing time is low because next to embeddings, these techniques generate distance scores out of the box.



(a) MUTAG dataset (b) BGS dataset

Fig. 10: Influence of substituting entities on the graph distances. We see that the remaining techniques all represent the gradual changes.

Table 3: Comparison of the different techniques for the various graph changes on the MUTAG dataset. We report the increase rate, i.e. the percentage of sequentially increasing distances. We also make a distinction between small (10) and larger (100) number of graph comparisons. We report the best rates in bolt.

	Add		Delete		Substitute		Entity		Schema	
#graphs:	10	100	10	100	10	100	10	100	10	100
GE-FSG	0.1370	0.0273	0.3167	0.3820	0.5515	0.1505	0.4407	0.0808	0.3333	**0.0529**
Graph2Vec	0.5985	0.2412	0.5967	0.3178	0.5067	0.2613	0.9925	0.2633	0.1112	0.0226
WL	0.9933	0.7172	**1.0**	0.9481	**0.8567**	**0.6910**	**1.0**	**1.0**	0.1112	0.0101
DGK	0.8766	0.4455	**1.0**	0.6566	0.5852	0.4926	0.9925	0.9882	0.4593	0.0875
GMN	**1.0**	**1.0**	**1.0**	**1.0**	0.7	0.6079	0.4445	0.1508	**0.5556**	0.0505
SIMGNN	0.1223	0.0212	0.2592	0.0329	0.3334	0.0384	/	/	/	/

Table 4: Comparison of the different techniques for the various graph changes on the BGS dataset. We report the increase rate, i.e. the percentage of sequentially increasing distances. We also make a distinction between small (10) and larger (100) number of graph comparisons. We report the best rates in bolt.

	Add		Delete		Substitute		Entity		Schema	
#graphs:	10	100	10	100	10	100	10	100	10	100
GE-FSG	0.4	0.02	0.3	0.1862	0.10	0.10	0.2222	0.1111	**0.3333**	0.0717
Graph2Vec	0.6699	0.25	0.6799	0.5140	0.6499	0.22	0.8888	0.6555	0.3222	0.0373
WL	0.99	0.9	**1.0**	0.9689	0.99	**0.82**	**1.0**	**1.0**	0.1111	0.0101
DGK	**1.0**	0.86	**1.0**	0.9565	**1.0**	0.79	0.8888	0.8333	**0.3333**	**0.0929**
GMN	**1.0**	**1.0**	**1.0**	**1.0**	0.99	0.76	0.8888	0.1666	0.2222	0.0444

5. Discussion

We will now discuss each technique separately:

Table 5: Time comparison (in seconds) of the different embedding techniques on the MUTAG dataset. We make a distinctions between generating the embeddings and comparing the embeddings to extract distances. Best results indicated in bolt.

	10 Graphs			100 Graphs		
	Total	Embedding	Comparing	Total	Embedding	Comparing
GE-FSG	23.9230	23.9179	0.0051	55.9493	55.4998	0.4495
Graph2Vec	2.2693	2.2641	0.0052	19.4116	18.9614	0.4502
WL	0.9338	0.8954	0.0383	68.4509	58.8058	9.6450
DGK	**0.8962**	**0.8927**	0.0034	**7.7421**	**7.4394**	0.3027
GMN	42.0052	42.0052	**0.0002**	60.7575	60.7572	**0.0002**
SimGNN	985.2861	985.2859	**0.0002**	1166.1246	1166.1245	**0.0002**

Table 6: Time comparison (in seconds) of the different embedding techniques on the BGS dataset. We make a distinctions between generating the embeddings and comparing the embeddings to extract distances. Best results indicated in bolt.

	10 Graphs			100 Graphs		
	Total	Embedding	Comparing	Total	Embedding	Comparing
GE-FSG	1.4074	1.4005	0.0069	2.6308	2.3541	0.2767
Graph2Vec	0.8978	0.8923	0.0054	4.7333	4.4585	0.2748
WL	**0.1528**	**0.1346**	0.0182	5.8411	3.5882	2.2528
DGK	0.3508	0.3474	0.0034	**2.2348**	**2.0451**	0.1896
GMN	23.3624	23.3623	**0.0002**	46.2661	46.2658	**0.0002**

5.1. *GE-FSG*

GE-FSG does not score well on any of the evaluations. Frequent subgraph matching, although promising in theory, does not work as well as path extraction techniques for graph comparison. While mining frequent subgraphs, frequently overlapping subgraphs in various graphs are detected. However, to calculate the distances between graphs, information regarding how the graphs differ is also very valuable. As GE-FSG mines frequent subgraphs, it focuses more on the reoccurring subgraphs and loses track of how the graphs differ, making it less suited to calculate the distances between graphs. Furthermore, as can be seen from Table 5 mining subgraphs is a lot more time consuming than extracting paths. Table 6 reports better results in terms of embedding times, as the graphs are smaller.

5.2. *WL*

WL excels in detecting added or deleted changes. This makes sense as this will result in different extracted WL-paths. This is clear when looking at the performance of the entity changes. There, WL scores best of all techniques, because when an entity changes, a different label during the WL-process will be assigned, leading to different paths. As WL basically counts to occurrences of the paths in each graph, it is easily able to detect these changes. Schema changes are hard to detect because

WL has no notion of a schema. It will just detect a new label, resulting in a different path. It handles the schema changes thus similar to entity changes. Table 5 shows, however, that the scalability of WL is bad. With an increasing number of graphs or larger graphs in general, the embedding time increases quickly. This is due to the complexity to construct the occurrence vectors, i.e. counting how many times each path occurs in each graph. Furthermore, due to the fact that WL is unable to produce fixed-sized vectors, it is less ideal for ML tasks.

5.3. *Graph2Vec*

Graph2Vec can detect most changes, however, it does not really excel in any of the tasks. Even though it uses WL-paths, it is able to detect the same kind of changes, however it performs worse. This is because Graph2Vec maps the paths to high dimensional embedding space, instead of counting the occurrences of the paths. We do see however, that the scalability of this approach is better for larger graphs and larger number of graphs. Furthermore, it produces fixed size embedding, making it ideal for ML tasks.

5.4. *DGK*

DGK performs really well on all evaluations (except the schema changes) while providing a very scalable solution, as it also relies on path extraction based on WL. The reason it does not perform well on the schema changes is the same as for WL. The combination with the graph kernel makes DKG much faster than plain WL in the time comparison. However, due to the fact that DGK is unable to produce fixed-sized vectors, it is less ideal for ML tasks.

5.5. *GMN*

On simple tasks such as 'Adding' and 'Deleting' GMN scores really well. Techniques such as GMN are best suited to learn the graph structure, compared to the walk-based methods that rather extract the graph content. As 'Adding' and 'Deleting' changes the structure of the graph itself, GMN can easily detect the changes. Substitution is harder, as the number of nodes and edges don't vary as much as while 'Adding' or 'Deleting'. This means that the structure of the graph stays rather constant, while the content changes. GMN is not suited to detect these kinds of changes. This is also clear in the 'Entity' evaluation, GMN scores rather low, because it does not, in the same magnitude, understand the content of the nodes in the graph.

5.6. *SimGNN*

Due to the bad performance and low scalability, we did not complete the full evaluation of SimGNN. The reason for the bad performance is the way SimGNN is trained.

It requires a labeled dataset with precomputed distances between the graphs in the training set. This causes two problems: 1) in practice, the distances between the graphs are not known; 2) the graphs in the training set should be a good representation of the graphs in the test set. This requires computing the GED for the graphs. Computing the GED is a very time-consuming task, which is one of the reasons to use embeddings in the first place.

5.7. *Summary*

We see that WL, DGK, Graph2Vec, and GMN are able to represent most of the KGs changes in their embedding space. WL suffers some scalability issues, while GMN is slow in general. However, none of the techniques are suited to take into account the schema information often found in KGs. This is mainly because the interpretation of the schema data is lost when converting the KGs to a more simple graph representation that can be used by the evaluated embedding techniques. In terms of usability for ML tasks, WL and DGK are not well suited, as they cannot produce fixed-length embeddings.

6. Conclusion

We have evaluated if whole KGs can be embedded through whole graph embedding and kernel techniques that were originally designed for heterogeneous graphs. We focused specifically on evaluating if similarities between KGs can be maintained and represented in the embedding space of these graph embedding techniques. Maintaining these similarities is important if we want to use whole KGs as input for ML algorithms, as the latter needs to be able to generalize over unseen data.

We have shown that when converting KGs to more simple graph representations such as heterogeneous graphs, techniques such as WL, Graph2Vec, DGK, and GMN are able to detect the most possible changes in KGs and represent these changes accordingly in their embedding space. However, as WL and DGK cannot produce fixed-length embeddings, they are less suited for ML tasks. Importantly, none of the techniques are able to incorporate the schema information often found in KGs.

For future work directions, solving this problem should be mandatory. Recently, efforts have started to solve this problem in the realm of KG node embeddings [12, 25]. Note that the focus of these efforts has been on embedding only the nodes of the graphs and not the whole graphs, as is the focus of this paper. Future work is necessary to evaluate if the techniques that are employed to inject the schema into node embeddings can provide a solution for creating schema-aware KG embeddings over the whole graph.

Acknowledgment: This research was funded by the imec.icon project RADIANCE, which was co-financed by imec, VLAIO, Barco, ML6 and Skyline. Pieter Bonte is funded by a postdoctoral fellowship of Fonds Wetenschappelijk Onderzoek Vlaanderen (FWO) (1266521N).

References

[1] F.M. Suchanek, G. Kasneci and G. Weikum, Yago: a core of semantic knowledge, in: *Proceedings of the 16th international conference on World Wide Web*, 2007, pp. 697–706.

[2] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak and Z. Ives, Dbpedia: A nucleus for a web of open data, in: *The semantic web*, Springer, 2007, pp. 722–735.

[3] K. Bollacker, C. Evans, P. Paritosh, T. Sturge and J. Taylor, Freebase: a collaboratively created graph database for structuring human knowledge, in: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, pp. 1247–1250.

[4] A. Singhal, Introducing the knowledge graph: things, not strings, *Official google blog* **16** (2012).

[5] M. Nickel, K. Murphy, V. Tresp and E. Gabrilovich, A review of relational machine learning for knowledge graphs, *Proceedings of the IEEE* **104**(1) (2015), 11–33.

[6] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang and S.Y. Philip, A comprehensive survey on graph neural networks, *IEEE Transactions on Neural Networks and Learning Systems* (2020).

[7] F. Bianchi, G. Rossiello, L. Costabello, M. Palmonari and P. Minervini, Knowledge Graph Embeddings and Explainable AI, *arXiv preprint arXiv:2004.14843* (2020).

[8] Y. Li, C. Gu, T. Dullien, O. Vinyals and P. Kohli, Graph Matching Networks for Learning the Similarity of Graph Structured Objects, in: *International Conference on Machine Learning*, 2019, pp. 3835–3845.

[9] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu and S. Jaiswal, graph2vec: Learning distributed representations of graphs, *arXiv preprint arXiv:1707.05005* (2017).

[10] P. Yanardag and S. Vishwanathan, Deep graph kernels, in: *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2015, pp. 1365–1374.

[11] H. Cai, V.W. Zheng and K.C.-C. Chang, A comprehensive survey of graph embedding: Problems, techniques, and applications, *IEEE Transactions on Knowledge and Data Engineering* **30**(9) (2018), 1616–1637.

[12] Y. Yu, Z. Xu, Y. Lv and J. Li, TransFG: A Fine-Grained Model for Knowledge Graph Embedding, in: *International Conference on Web Information Systems and Applications*, Springer, 2019, pp. 455–466.

[13] P. Ristoski, J. Rosati, T. Di Noia, R. De Leone and H. Paulheim, RDF2Vec: RDF graph embeddings and their applications, *Semantic Web* **10**(4) (2019), 721–752.

[14] M. Nickel, L. Rosasco and T. Poggio, Holographic embeddings of knowledge graphs, in: *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 30, 2016.

[15] G.K.D. de Vries and S. de Rooij, Substructure counting graph kernels for machine learning from RDF data, *Journal of Web Semantics* **35** (2015), 71–84. doi:10.1016/j.websem.2015.08.002. `https://linkinghub.elsevier.com/retrieve/pii/S1570826815000657`.

[16] X. Gao, B. Xiao, D. Tao and X. Li, A survey of graph edit distance, *Pattern Analysis and applications* **13**(1) (2010), 113–129.

[17] D.B. Blumenthal and J. Gamper, On the exact computation of the graph edit distance, *Pattern Recognition Letters* **134** (2020), 46–57.

[18] D.L. McGuinness, F. Van Harmelen et al., OWL web ontology language overview, *W3C recommendation* **10**(10) (2004), 2004.

[19] N.M. Kriege, F.D. Johansson and C. Morris, A survey on graph kernels, *Applied Network Science* **5**(1) (2020), 1–42.

[20] N. Shervashidze, P. Schweitzer, E.J.v. Leeuwen, K. Mehlhorn and K.M. Borgwardt, Weisfeiler-lehman graph kernels, *Journal of Machine Learning Research* **12**(Sep) (2011), 2539–2561.

[21] T. Mikolov, I. Sutskever, K. Chen, G.S. Corrado and J. Dean, Distributed Representations of Words and Phrases and their Compositionality, in: *Advances in Neural Information Processing Systems 26*, C.J.C. Burges, L. Bottou, M. Welling, Z. Ghahramani and K.Q. Weinberger, eds, Curran Associates, Inc., 2013, pp. 3111–3119.

[22] Q. Le and T. Mikolov, Distributed representations of sentences and documents, in: *International conference on machine learning*, 2014, pp. 1188–1196.

[23] D. Nguyen, W. Luo, T.D. Nguyen, S. Venkatesh and D. Phung, Learning graph representation via frequent subgraphs, in: *Proceedings of the 2018 SIAM International Conference on Data Mining*, SIAM, 2018, pp. 306–314.

[24] T.N. Kipf and M. Welling, Semi-supervised classification with graph convolutional networks, *arXiv preprint arXiv:1609.02907* (2016).

[25] Ö. Özçep, M. Leemhuis and D. Wolter, Cone semantics for logics with negation, in: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, 2020.

[26] G.K.D. De Vries, LNAI 8188 - A Fast Approximation of the Weisfeiler-Lehman Graph Kernel for RDF Data, Technical Report. `http://www.openrdf.org/`.