J Inf Process Syst, Vol.16, No.6, pp.1407~1423, December 2020
https://doi.org/10.3745/JIPS.04.0197

ISSN 1976-913X (Print)
ISSN 2092-805X (Electronic)

JOURNAL OF INFORMATION PROCESSING SYSTEMS JIPS

# A Graph Embedding Technique for Weighted Graphs Based on LSTM Autoencoders

Minji Seo* and Ki Yong Lee*

**Abstract**
A graph is a data structure consisting of nodes and edges between these nodes. Graph embedding is to generate a low dimensional vector for a given graph that best represents the characteristics of the graph. Recently, there have been studies on graph embedding, especially using deep learning techniques. However, until now, most deep learning-based graph embedding techniques have focused on unweighted graphs. Therefore, in this paper, we propose a graph embedding technique for weighted graphs based on long short-term memory (LSTM) autoencoders. Given weighted graphs, we traverse each graph to extract node-weight sequences from the graph. Each node-weight sequence represents a path in the graph consisting of nodes and the weights between these nodes. We then train an LSTM autoencoder on the extracted node-weight sequences and encode each node-weight sequence into a fixed-length vector using the trained LSTM autoencoder. Finally, for each graph, we collect the encoding vectors obtained from the graph and combine them to generate the final embedding vector for the graph. These embedding vectors can be used to classify weighted graphs or to search for similar weighted graphs. The experiments on synthetic and real datasets show that the proposed method is effective in measuring the similarity between weighted graphs.

**Keywords**
Graph Embedding, Graph Similarity, LSTM Autoencoder, Weighted Graph Embedding, Weighted Graph

## 1. Introduction

A graph is a data structure consisting of nodes and edges between these nodes. Graphs are widely used to represent data in various fields such as chemistry, biology, and social networking services (SNS). Recently, with the advance of deep learning techniques [1-3], deep learning-based graph analysis has been conducted on various topics. Among them, graph embedding is to represent a graph as a vector in a low-dimensional space. The goal of graph embedding is to generate a low dimensional vector for a given graph that best represents the characteristics of the graph. Thus, if two graphs are similar, their embedding vectors should be similar.

In recent years, as the use of deep learning has increased in various fields including image recognition and natural language processing, research on graph embedding using deep learning has also been actively conducted [4,5]. Deep learning-based graph embedding methods use various deep learning models and techniques to extract the features of graphs and encode the extracted features into fixed-length vectors.

※ This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.org/licenses/by-nc/3.0/) which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.

Manuscript received March 19, 2020; first revision June 1, 2020; accepted June 7, 2020.

**Corresponding Author:** Ki Yong Lee (kiyonglee@sookmyung.ac.kr)
* Dept. of Computer Science, Sookmyung Women's University, Seoul, Korea (minkky@sookmyung.ac.kr, kiyonglee@sookmyung.ac.kr)



Copyright© 2020 KIPS

However, until now, most deep learning-based graph embedding methods have focused on the structure of graphs (i.e., topology or connections between nodes) and do not take much into account the weights of edges in graphs. As a result, embedding vectors generated by most existing deep learning-based graph embedding methods do not contain the information about the weights of edges in graphs.

A weighted graph is a graph in which each edge is assigned a weight. In real applications, weighted graphs are widely used to represent chemical compounds, where each weight represents the distance or bond strength between atoms, and social networks, where each weight represents the degree of closeness between people. Consider the three chemical compounds shown in Fig. 1. Fig. 1(a) and (b) have the same structure but very different weights, while Fig. 1(a) and (c) have the same structure and similar weights. If we apply most of the existing deep learning-based graph embedding methods to the three graphs in Fig. 1, similar embedding vectors are generated for all the three graphs because these methods do not consider the weight of edges in the graphs. However, considering the weights, the embedding vectors of Fig. 1(a) and (c) should be more similar than those of Fig. 1(a) and (b). This can be a very serious problem when we deal with weighted graphs because the weights of edges is one of the important characteristics of graphs.

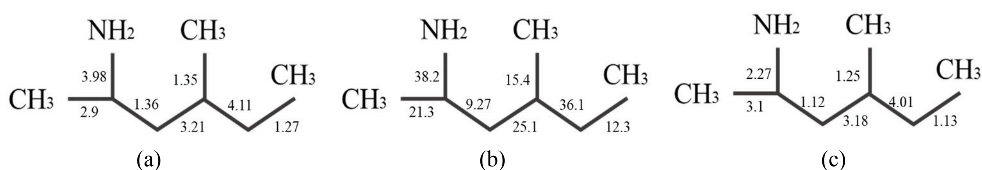

**Fig. 1.** Examples of graphs with different weights.

Therefore, in this paper, we propose a graph embedding technique for weighted graphs based on long short-term memory (LSTM) autoencoders. Given weighted graphs, we traverse each graph in a pre-defined way to extract node-weight sequences from the graph. A node-weight sequence of a graph is a sequence of nodes and weights in the graph, listed in the order in which they are visited, starting from a particular node in the graph. Each node-weight sequence extracted from a graph can be considered to represent partial information about the structure and weights of the graph. After extracting node-weight sequences from all the given weighted graphs, we train an LSTM autoencoder on the extracted node-weight sequences and encode each node-weight sequence into a fixed-length vector using the trained LSTM autoencoder. Finally, for each graph, we collect the encoding vectors obtained from the graph and combine them to generate the final embedding vector for the graph. The embedding vectors generated in this way include not only information about the structure of the graphs, but also information about the weights of the graphs. These embedding vectors can be used to classify weighted graphs or to search for similar weighted graphs. The experiments on synthetic and real datasets show that the proposed method is very effective in measuring the similarity between weighted graphs.

The remainder of the paper is organized as follows. In Section 2, we present related work on graph embedding and describe the deep learning model used in the proposed method. Section 3 describes the proposed graph embedding technique in detail. In Section 4, we present ~~the~~ experimental results on synthetic and real datasets to show the effectiveness of the proposed method. Finally, we conclude in Section 5.

# 2. Related Work

## 2.1 Graph Embedding

The goal of graph embedding is to represent a given graph as a low-dimensional vector while maintaining the characteristics of the graph as much as possible. According to how to get the representation of graphs, graph embedding methods can be largely classified into three categories [5]: matrix factorization methods, graph kernel methods, and deep learning-based methods.

(1) Matrix factorization methods: Early methods of graph embedding primarily attempted to solve graph embedding problems using matrix factorization [4,5]. These methods represent the property of a graph in the form of a matrix and factorize this matrix to obtain its embedding vector.

(2) Graph kernel methods: These methods represent each graph as a vector according to a predetermined scheme (e.g., by expressing sub-graphs or sub-tree patterns, or by conducting random-walks) and then measure the similarity between two graphs by calculating the inner product of their representing vectors [6].

(3) Deep learning-based methods: These methods apply existing deep learning models to extract the features of graphs or develop a new deep learning model specialized in extracting the features of graphs. In this paper, we focus on deep learning-based methods because they do not require manual selection of features and show good performance in many applications [7-11]. Let us describe deep learning-based methods in more detail below.

Deep learning-based graph embedding methods are divided into two types: node embedding and whole-graph embedding. Node embedding is to learn a low-dimensional vector representation of each node in a graph, whereas whole-graph embedding is to represent the whole graph as a single vector. In this paper, we focus on whole-graph embedding.

These two studies, [7] and [8], are representative node embedding methods based on deep learning. Scarselli et al. [7] propose the graph neural network (GNN), which is a neural network model used to represent graph structures. GNN extends the recurrent neural network (RNN) to learn the representation of each node. Given the structure information of a graph (e.g., the degree of each node in the graph), the GNN generates the embedding vector of each node by using the feature vector representing the structure information of that node and the feature vectors of its neighboring nodes. To obtain the embedding vector of a node $N$, the feature vector of $N$ and the feature vectors of the neighboring nodes of $N$ are aggregated. This process is repeated until the embedding vectors of all nodes become stable. Kipf and Welling [8] propose the graph convolutional network (GCN), which is a variant of the method in [7]. The authors [8] use a different way to aggregate the structure information of neighboring nodes when generating the embedding vector of a node. Unlike [7] that uses the average when aggregating the structure information of neighboring nodes, [8] assigns a different weight to each neighbor based on its degree when aggregating their structure information.

Several studies [9–11] are representative whole-graph embedding methods based on deep learning. Teheri et al. [9] extract node sequences from given graphs by traversing the graphs and listing their nodes in order of visit. To traverse a graph, [9] uses several algorithms such as random walk or the all-pairs shortest-path algorithm. It then uses the node sequences to train an RNN autoencoder and encodes those sequences into fixed-length vectors using the trained RNN autoencoder. Finally, the final embedding vector of each graph is obtained by averaging the encoding vectors of the node sequences extracted from

the graph.

Yanardag and Vishwanathan [10] divide graphs into their sub-structures (e.g., sub-graphs, sub-tree patterns, or random-walk sequences), trains a deep learning model on their sub-structures, and then generates their embedding vectors using the trained deep learning model. Because the number of sub-structures of graphs increases rapidly as the size of the graphs increases, [10] uses a deep learning model to learn the characteristics of the sub-structures of the graphs effectively. When training the deep learning model, it uses the edit distance to measure the similarity between sub-structures or between graphs.

Narayanan et al. [11] apply doc2vec [12] to graph embedding, where doc2vec is an embedding method that extends word2vec [13] to document embedding. As a document can be considered as a set of words, a graph can be seen as a set of subgraphs. The authors [11] propose the graph2vec model that divides each graph into subgraphs and trains a neural network on the subgraphs, whose goal is to maximize the probability that subgraphs belonging to the same graph appear together. When training the neural network, it represents each subgraph as a one-hot vector and uses a skip-gram model. Once the model is trained, each graph is input into the model and the values of the hidden layer of the model become the embedding vector of the graph.

In our previous work [14], we presented preliminary results of using an LSTM autoencoder for weighted graph embedding. Compared with [14], we provide a more comprehensive study and analysis of a graph embedding technique for weighted graphs based on LSTM autoencoders. In this paper, we investigate the effects of various node encoding schemes, various loss functions of LSTM autoencoders, and various final embedding vector generation methods.

However, except [14], most existing deep learning-based whole graph embedding methods, including [9], [10], and [11], do not consider the weights of edges in graphs, but only consider the structure information of graphs (i.e., connections between nodes) to generate embedding vectors.

## 2.2 Deep Learning Models

This section briefly describes the deep learning model used in this paper, i.e., an LSTM autoencoder, before presenting the proposed method.

### 2.2.1 Long short-term memory

LSTM) [15] is a neural network architecture proposed to overcome the vanishing gradient problem of the traditional RNN. The vanishing gradient problem is that as new data continue to arrive, the influence of old data decreases drastically. Because of this problem, the traditional RNN has a disadvantage that it does not have long-term memory and predicts the next data by considering only recently arrived data. LSTM solves this vanishing gradient problem by adding a cell state to each unit in the network. Furthermore, each unit is extended with an input gate, a forget gate, and an output gate. In Fig. 2, $x_t$ and $y_t$ represent the input and output at time $t$, and the output of a hidden node at time $(t - 1)$ comes back as its input at time $t$. The forget gate determines how much of the previous state is retained and the input gate determines how much of the current input will be used. The output gate determines how much of the current state is passed out. LSTM uses these gate structures and cell state to selectively store information from previous states. As a result, LSTM can solve the vanishing gradient problem and shows better performance because it uses old data to predict the next data.

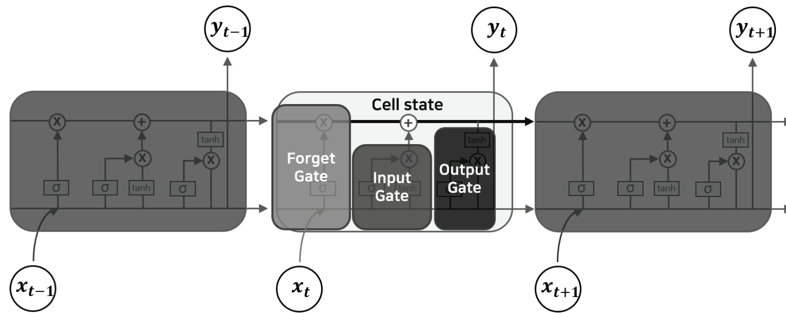


**Fig. 2.** The architecture of LSTM.

### 2.2.2 Autoencoder

An autoencoder [16] is a representative neural network architecture for unsupervised learning, whose goal is to learn the hidden representation of data. As shown in Fig. 3, an autoencoder has the same number of nodes in its input and output layers and a smaller number of nodes in its hidden layers. An autoencoder is trained with the purpose of reconstructing its input, i.e., minimizing the difference between the input ($x_1$, $x_2$, $x_3$, $x_4$, $x_5$) and the output ($y_1$, $y_2$, $y_3$, $y_4$, $y_5$). Because the number of nodes in hidden layers is smaller than those in the input layer, it is possible to obtain compressed or noise-removed data from the hidden layers [17]. An autoencoder is mainly composed of an encoder and a decoder. The encoder plays the role of extracting the features of the input data, and the decoder plays the role of reconstructing the input data from the extracted features. As a result, once the training of the autoencoder is completed, the features of the input data can be obtained from the bottleneck layer, which is located in the middle of the model.

In the proposed method, we extract node-weight sequences from given graphs and train an LSTM autoencoder on those sequences. An LSTM autoencoder is an autoencoder for sequence data, where the encoder and decoder have the LSTM architecture. Once the LSTM autoencoder is trained, we can obtain the encoding vector of each node-weight sequence by inputting it into the LSTM autoencoder and taking the values of the bottleneck layer.

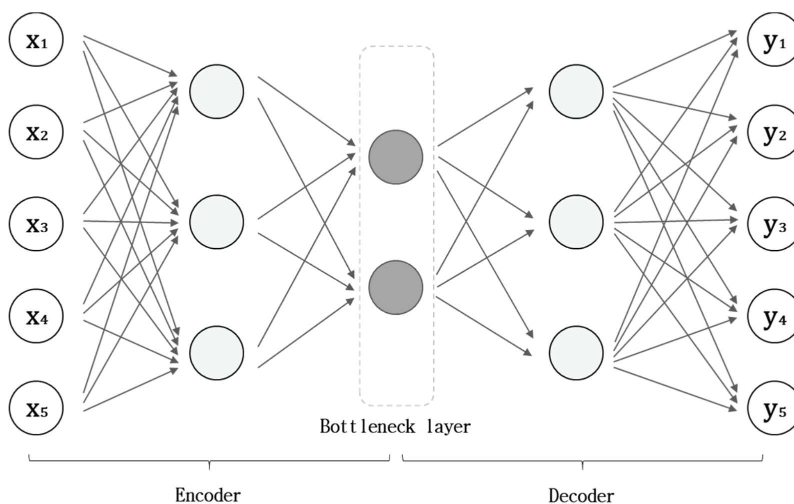


**Fig. 3.** The architecture of an autoencoder.

# 3. Proposed Method

In this section, we present the proposed graph embedding technique for weighted graphs. The proposed graph embedding technique consists of three steps: node-weight sequence extraction, node-weight sequence encoding, and final embedding vector generation. We now describe each step in detail.

## 3.1 Node-Weight Sequence Extraction

The purpose of the proposed method is to generate embedding vectors for given weighted graphs that represent both the structure and weight information of the graphs. Given weighted graphs, the proposed method first extracts node-weight sequences from each graph. A node-weight sequence of a graph is a sequence of nodes and weights in the graph, listed in the order in which they are visited, starting from a specified node in the graph. Fig. 4 shows an example of a weighted graph and a node-weight sequence extracted from the graph, which starts from node *D*.

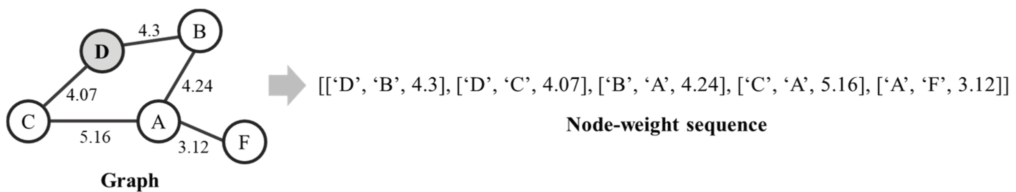


**Fig. 4.** An example of extracting a node-weight sequence from a graph.

Let $G_1$, $G_2$, ⋯, $G_N$ be weighted graphs to be embedded. Unlike the previous methods, we first extract node-weight sequences from each $G_i$ ($i$ = 1, 2, ⋯, $N$), which contain both the structure and weight information of $G_i$. Let $N_i$ be the set of nodes in $G_i$. For each node $n_{(1)} \in N_i$, the proposed method traverses all nodes of $G_i$ starting from $n_{(1)}$ to extract a node-weight sequence of $G_i$. In this paper, we use the breadth-first search (BFS) algorithm to traverse all nodes of $G_i$. We then generate a sequence listing all nodes and weights in their order of visit, which is expressed as $[[n_{(1)}, n_{(2)}, w_{(1)}], [n_{(2)}, n_{(3)}, w_{(2)}], \cdots, [n_{(|G_i|-1)}, n_{(|G_i|)}, w_{(|G_i|-1)}]]$, where $|G_i|$ denotes the number of nodes in $G_i$. Here, $n_{(1)}, \cdots, n_{(|G_i|)}$ represent nodes listed in order of visit according to the BFS algorithm, and $w_{(i)}$ represents the weight of the edge between $n_{(i)}$ and $n_{(i+1)}$. Through this process, a total of $|G_i|$ node-weight sequences are extracted from $G_i$, each of which starts at each node in $G_i$. Therefore, for given graphs $G_1$, $G_2$, ⋯, $G_N$, a total of $|G_1| + |G_2| + \cdots + |G_N|$ node-weight sequences are obtained.

Note that we use the BFS algorithm to extract node-weight sequences from a graph. The BFS algorithm first visits the nodes close to the starting node and then visits the distant nodes later. It is known that if the neighbors extracted by the BFS algorithm are similar, there is a structural equivalence between the graphs [18]. Also, it is known that the order of nodes visited by the BFS algorithm represents the structure of the graph very well [18].

## 3.2 Node-Weight Sequence Encoding

Once we extract node-weight sequences from $G_1$, $G_2$, ⋯, $G_N$, the next step is to encode the extracted node-weight sequences into fixed-length vectors. For this purpose, we train an LSTM autoencoder on the

extracted node-weight sequences. We then use the trained LSTM autoencoder to encode each node-weight sequence into a fixed-length vector. However, in order to input a node-weight sequence into an LSTM autoencoder, it is necessary to encode each node label into a number. In this paper, we use the ordinal encoding method and the one-hot encoding method, respectively. The ordinal encoding method lists the labels of nodes in a certain order and encodes each label into its order in the list. For example, if we are given the labels of nodes 'A', 'B', …, 'Z', then we can list them in alphabetical order and encode 'A,', 'B', …, 'Z' into 1, 2, …, 26, respectively. When the number of node labels is $L$, the one-hot encoding method encodes each node label as an $L$-dimensional 0-1 vector where only the corresponding element is 1. For example, in the above case, we can encode 'A', 'B', …, 'Z' into [1, 0, 0, …, 0], [0, 1, 0, …, 0], …., [0, 0, 0, …, 1], respectively, where each vector is 26-dimensional.

After we encode node labels using either of the two methods, we train an LSTM autoencoder on node-weight sequences. Fig. 5 shows the architecture of the LSTM autoencoder used in the paper. The LSTM autoencoder we used in this paper consists of 7 layers. Like a conventional autoencoder, the LSTM autoencoder we used consists of an encoder, a decoder, and a bottleneck layer. The top 3 layers are the encoder layers, which receive a node-weight sequence of arbitrary length, extract its features, and encode them into a 64-dimensional vector. The fourth layer is the bottleneck layer, which plays the role of converting a 64-dimensional vector into a sequence form to prepare the reconstruction of the original node-weight sequence. The bottom 3 layers are the decoder layers, which reconstruct the original node-weight sequence from the output of the fourth layer.

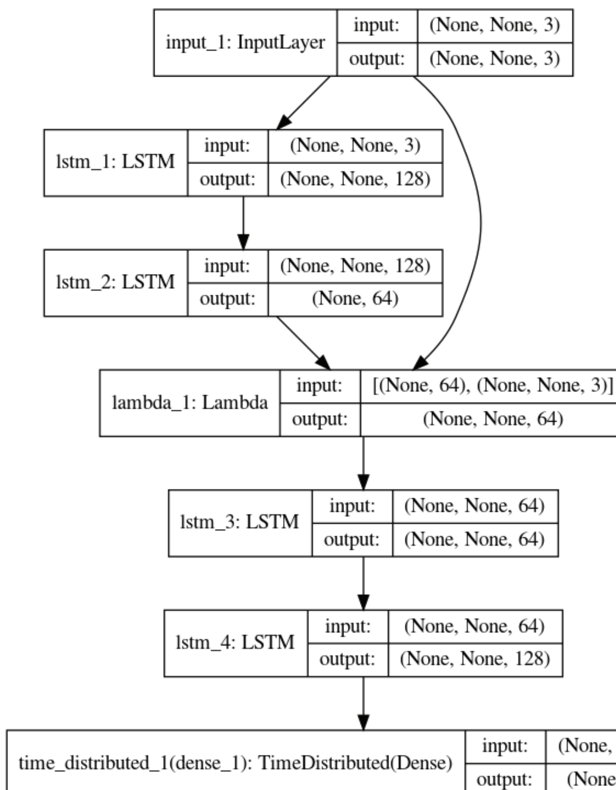


**Fig. 5.** The architecture of the LSTM autoencoder used for node-weight sequence encoding.

In Fig. 5, the arrows between layers represent the direction of data flow. The left block in each layer shows the name of that layer and the right block shows the shapes of input and output data of that layer, where the first, second, and third elements of a shape represent the batch size, the number of data, and the dimension of data, respectively. The top layer has the name "input_1" and its input and output have the shape of (None, None, 3), which means that the batch size and the length of a sequence are not fixed and each sequence consists of 3-dimensional vectors (i.e., $[n_{(i)}, n_{(i+1)}, w_{(i)}]$). The next two layers, named "lstm_1" and "lstm_2", correspond to the encoder layers, which have the LSTM architecture. The layer "lstm_1" compresses an input sequence into a 128-dimensional vector and the layer "lstm_2" compresses a 128-dimensional vector into a 64-dimensional vector. The layer named "lambda_1" converts a 64-dimensional vector into a sequence form by repeating the values of the vector to prepare the reconstruction of the original sequence. The next two layers, named "lstm_3" and "lstm_4", correspond to the decoder layers, which have the LSTM architecture. The layer "lstm_3" reconstructs the 64-dimensional vector from the output of the layer "lambda_1" and the layer "lstm_4" reconstructs the 128-dimensional vector from the output of the layer "lstm_3". Finally, the last layer, named "time_distributed_1", is a fully-connected layer and plays the role of reconstructing the original sequence from the output of the layer "lstm_4".

Using the LSTM autoencoder described above, we can encode each node-weight sequence with a different length into a fixed-length vector. To do this, we train the LSTM autoencoder on all the extracted node-weight sequences. Then, we can obtain the encoding vector of each node-weight sequence by inputting it into the trained LSTM autoencoder and extracting the output of the layer "lstm_2", which is a 64-dimensional vector. Each encoding vector represents the characteristics of the corresponding node-weight sequence.

When we train an LSTM autoencoder, we need to define its loss function, which represents the difference between the input and output of the LSTM autoencoder. The goal of training an LSTM autoencoder is to minimize the difference between its input and output. In this paper, we use a combination of the following three loss functions to train the LSTM autoencoder:

- **Mean squared error (MSE):** MSE is one of the most widely used measures for the difference between input and output, which is defined as follows:

$$\mathrm{MSE} = \frac{1}{m}\sum_{i=1}^{m}({y_{true}}_i - {y_{pred}}_i)^2 \tag{1}$$

  In Eq. (1), *m* is the number of training data, ${y_{true}}_i$ is the correct output for the *i*th training data, and ${y_{pred}}_i$ is the output of the model for the *i*th training data. In our case, ${y_{true}}_i$ is the *i*th node-weight sequence and ${y_{pred}}_i$ is the node-weight sequence reconstructed by the LSTM autoencoder for ${y_{true}}_i$. We used MSE as the first loss function.

- **Kullback-Leibler Divergence (KLD):** KLD is a metric used to measure the difference between two probability distributions, which is defined as follows:

$$\mathrm{KLD} = D_{KL}(P||Q) = \sum_{i} P(i)\, log\frac{P(i)}{Q(i)} \tag{2}$$

  In Eq. (2), $P$ and $Q$ are the distributions of the correct outputs and the outputs of the model,

respectively. $P(i)$ and $Q(i)$ are the probability of the $i$th correct output and the $i$th output of the model to occur, respectively. We used the sum of MSE and KLD, denoted by MSE+KLD, as the second loss function.

- **Categorical cross-entropy (CCE):** CCE is a measure of the difference between two probability distributions when the output is represented by a one-hot vector. CCE is defined as follows:

$$\text{CCE} = -\sum_{i} P(i) \log\left(Q(i)\right) \tag{3}$$

In Eq. (3), $P$, $Q$, $P(i)$, and $Q(i)$ have the same meanings as in Eq. (2). CCE is frequently used for a model where the output layer is a softmax layer. We used the sum of MSE and CCE, denoted by MSE+CCE, as the third loss function.

In this paper, we used the three loss functions (i.e., MSE, MSE+KLD, and MSE+CCE) to train the LSTM autoencoder. We present the effect of using the three loss functions in Section 4.

## 3.3 Final Embedding Vector Generation

Once we obtain the encoding vectors for all the node-weight sequences, which are extracted from $G_1$, $G_2$, ⋯, $G_N$, we can generate the final embedding vector $V_i$ for each $G_i$ ($i$ = 1, 2, ⋯, $N$). For a graph $G_i$, let $v_1, v_2, \cdots, v_{|G_i|}$ be the encoding vectors of the node-weight sequences extracted from $G_i$. We can think of each $v_i$ as representing partial information about the structure and weights of $G_i$. Thus, there can be many ways to combine $v_1, v_2, \cdots, v_{|G_i|}$ to generate the final embedding vector $V_i$ for $G_i$. In this paper, we use three methods to generate $V_i$ from $v_1, v_2, \cdots, v_{|G_i|}$, whose goal is to preserve the information contained in $v_1, v_2, \cdots, v_{|G_i|}$ as much as possible.

- **Mean vector:** $V_i$ is generated as the mean vector of $v_1, v_2, \cdots, v_{|G_i|}$, which can be expressed as follows:

$$V_i = \frac{1}{|G_i|}\sum_{i=1}^{|G_i|} v_i \tag{4}$$

- **Trimmed mean vector:** $V_i$ is generated as the trimmed mean vector of $v_1, v_2, \cdots, v_{|G_i|}$, which can be expressed as follows:

$$V_{i,j} = \frac{1}{|G_i| - 2}\left\{\sum_{i=1}^{|G_i|} v_{i,j} - \max\left\{v_{1,j}, \dots, v_{|G_i|,j}\right\} - \min\left\{v_{1,j}, \dots, v_{|G_i|,j}\right\}\right\} \tag{5}$$

In Eq. (5), $V_{i,j}$ and $v_{i,j}$ represent the $j$th element in $V_i$ and $v_i$, respectively ($j$ = 1, 2, …, 64). In other words, $V_{i,j}$ is the average of $v_{1,j}, \dots, v_{|G_i|,j}$ excluding their maximum and minimum values. This method can eliminate the effect of outliers among $v_{1,j}, \dots, v_{|G_i|,j}$.

- **Mode vector:** $V_i$ is generated as the mode vector of $v_1, v_2, \cdots, v_{|G_i|}$, which can be expressed as follows:

$$V_{i,j} = most_frequent_value \left\{ v_{1,j}, \dots, v_{|G_i|,j} \right\} \tag{6}$$

In Eq. (6), the function *most_frequent_value* returns the most frequent value among $v_{1,j}, \dots, v_{|G_i|,j}$. If there are multiple most frequent values, *most_frequent_value* returns their average value.

In Section 4, we present the performance of the proposed method for each of the above three methods, respectively.

# 4. Experiments

## 4.1 Experimental setting

In order to confirm whether the proposed method actually generates similar embedding vectors for similar weighted graphs, we evaluated the performance of the proposed method using synthetic and real datasets. For the synthetic dataset, we defined three graphs A, B, and C shown in Fig. 6, their shapes being derived from the actual compounds (i.e., Propane, Benzene, and Tetrahedrane, respectively). For each of the three graphs, we created two groups of graphs that are similar in shape to the graph, but with significantly different weight ranges. As a result, a total of six groups of graphs were created, as shown in Table 1. We synthetically generated 100 graphs for each group by randomly selecting one or two nodes from the group's base graph (i.e., A, B, or C) and randomly performing one of the following operations: (1) create and connect a new node, (2) delete the node, (3) modify the label of the node, or (4) modify the weight of an edge connected to the node slightly (±10%). As a result, graphs in the same group have similar shapes and weights, while graphs in different groups have different shapes or weights.

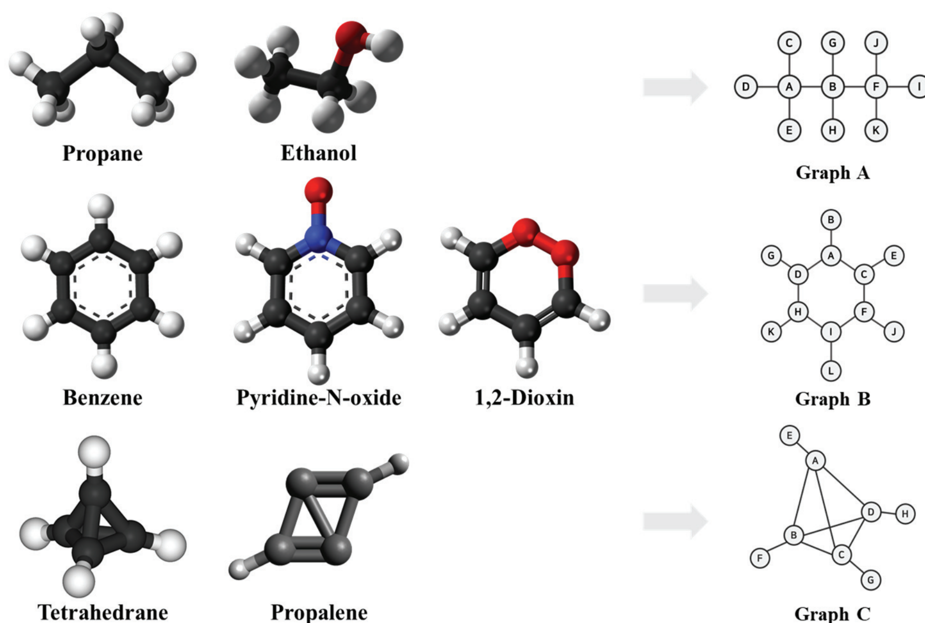


**Fig. 6.** The shapes of the three graphs used to generate the synthetic dataset.

**Table 1.** Synthetic graph groups used in the experiment

| Group | Description | Weight range |
|---|---|---|
| 1 | Graphs similar in shape to Graph A | 0.0–30.0 |
| 2 | | 50.0–185.0 |
| 3 | Graphs similar in shape to Graph B | 0.0–30.0 |
| 4 | | 50.0–185.0 |
| 5 | Graphs similar in shape to Graph C | 0.0–30.0 |
| 6 | | 50.0–185.0 |

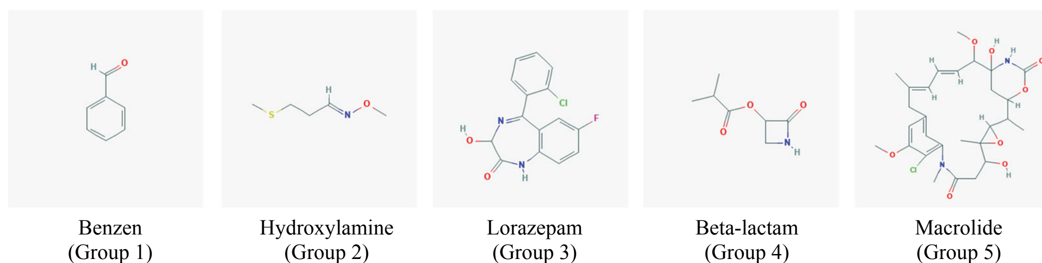


**Fig. 7.** The five real chemical compounds used in the experiments.

For the real dataset, we used the real chemical compound dataset provided by PubChem [19]. PubChem is a large database provided and managed by the National Center for Biotechnology Information (NCBI), which contains structures, descriptions, and experimental results for more than 100 million chemicals. From PubChem, we selected five compounds (i.e., Benzene, Hydroxylamine, Lorazepam, Beta-lactam, and Macrolide), which are shown in Fig. 7. Then, for each compound, we selected 20 similar compounds with the help of experts. Thus, in this case, a total of five graph groups were created.

In the experiments, we evaluated whether the proposed graph embedding technique generates similar embedding vectors for similar weighted graphs and different embedding vectors for different weighted graphs. For this purpose, we conducted the experiments as follows: given groups of weighted graphs, we first generated the embedding vectors of all the graphs in the groups using a graph embedding technique to be evaluated. After that, we randomly selected a fixed number of graphs from each group. Then, for each selected graph, we extracted the *k* graphs with embedding vectors closest to the graph's embedding vector among all the graphs in the dataset. We then measured the precision at *k*, which is the proportion of graphs belonging to the same group as the selected graph among the extracted *k* graphs. Therefore, the higher the value of precision at *k*, the better the embedding technique generates similar vectors for similar weighted graphs and different vectors for different weighted graphs. We used the cosine distance to measure the distance between two embedding vectors.

In the experiments, we compared the proposed method with the methods proposed in [9] and [11], denoted by RNN autoencoder and Graph2vec, respectively. For the proposed method, we used the ordinal and one-hot encoding to encode node labels, respectively, and trained the LSTM autoencoder with the loss functions MSE, MSE+KLD, and MSE+CCE, respectively. Also, we used the mean, trimmed mean, and mode vectors to generate final embedding vectors, respectively. In Section 4.2, we present the performance of the proposed method for each case. For RNN autoencoder and Graph2vec, we set all their parameters to the values used in [9] and [11], respectively.

We implemented all the methods in Python 3.7.4 using the Jupyter Notebook 6.0.1, and built an LSTM

autoencoder using Keras 2.2.5 and TensorFlow GPU 1.14.0. All the experiments were performed on a PC running Ubuntu 16.04.6 LTS equipped with Intel Core i7-9700K 3.60 GHz CPU, 16 GB RAM, 1 TB HDD, and NVIDIA Titan V GPU. We released our code on GitHub at https://github.com/minkky/Graph-Embedding.

## 4.2 Evaluation Results

### 4.2.1 Performance evaluation on synthetic dataset

In this section, we present the performance evaluation results on the synthetic dataset. In this experiment, we first generated the embedding vectors of all the graphs in the synthetic dataset using the proposed graph embedding technique. It took about 3 hours and 40 minutes to extract node-weight sequences, train an LSTM autoencoder, and generate the final embedding vectors of all the graphs. The total number of node-weight sequences extracted from all the graphs was 4,200. We then randomly selected 10 graphs from each of the six groups in the synthetic dataset. For each selected graph, we extracted the $k$ graphs with the closest embedding vectors, which were generated by the proposed method, and then measured precision at $k$. We measured precision at $k$ by increasing $k$ from 10 to 50. Fig. 8 shows the performance of the proposed method on the synthetic dataset. Fig. 8(a), (b), and (c) show precision at $k$ when node labels are encoded using the ordinal encoding, while Fig. 8(d) and (e) show precision at $k$ when node labels are encoded using the one-hot encoding. Also, Fig. 8(a) and (d), Fig. 8(b) and (e), and Fig. 8(c) show precision at $k$ when the loss function is MSE, MSE+KLD, and MSE+CCE, respectively (Note that we do not present precision at $k$ when node labels are encoded using the one-hot encoding and the loss function is MSE+CCE because the LSTM autoencoder failed to be trained in that case, i.e., the loss function diverged). In each subfigure of Fig. 8, we compared precision at $k$ when using the mean, trimmed mean, and mode vectors to generate final embedding vectors.

In Fig. 8, we can see that embedding vectors generated by the proposed method always show more than 95% precision at $k$. This means that the proposed method correctly generates similar embedding vectors for similar weighted graphs and dissimilar embedding vectors for graphs with different structures or different weights. Therefore, we can confirm that the proposed method is very effective for embedding weighted graphs.

When comparing the ordinal and one-hot encoding, the one-hot encoding shows slightly better performance than the ordinal encoding, especially when the mean or trimmed mean vectors are used to generate final embedding vectors. This is because, when training the LSTM autoencoder with node labels encoded by the ordinal encoding, nodes with similar encoding values may be misinterpreted as similar nodes. For example, if node labels “A”, “B”, and “C” are encoded as 1, 2, and 3, respectively, by the ordinal encoding, “A” and “B” may be misinterpreted as being more similar than “A” and “C”. Thus, by being trained in this way, the LSTM autoencoder may learn wrong features from node-weight sequences. On the other hand, this problem is alleviated when we use the one-hot encoding.

When comparing the three loss functions, MSE+KLD and MSE+CCE generally show better performance than MSE. This means that when training the LSTM autoencoder, it is more effective to consider not only the difference between each input and output, but also the difference between the distributions of inputs and outputs.

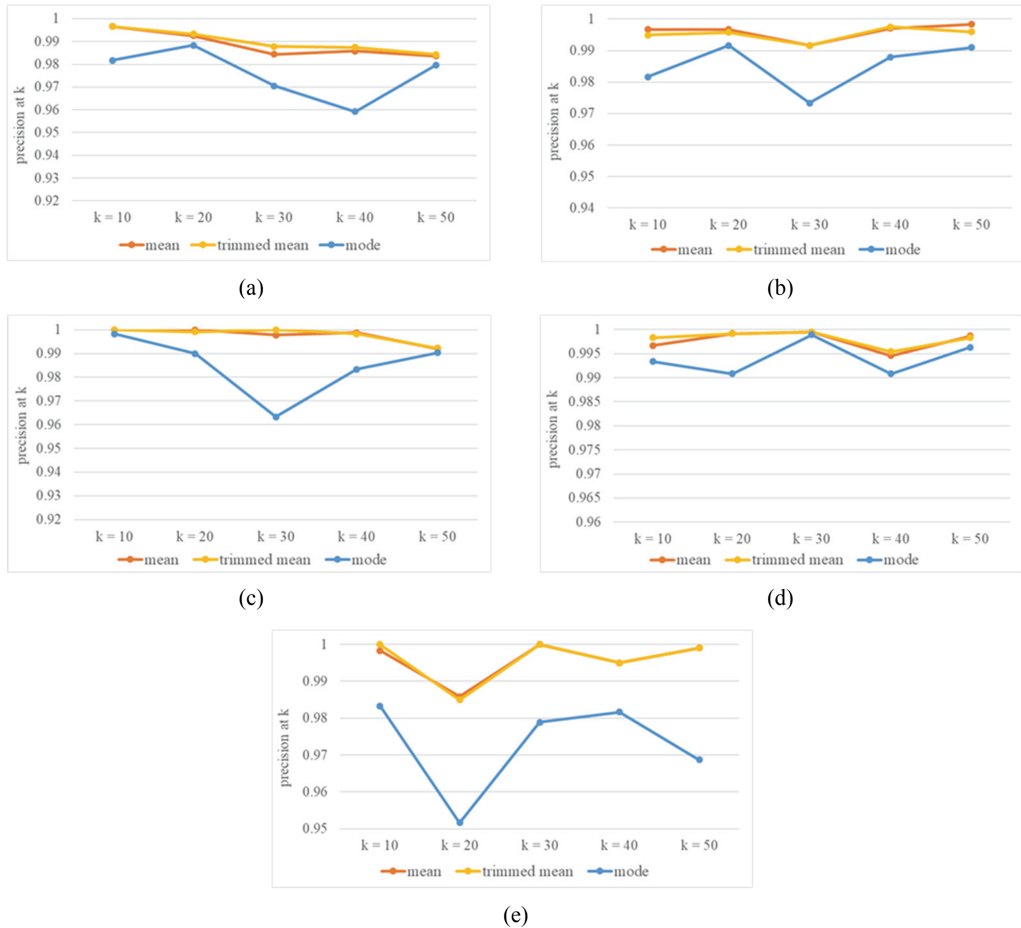


**Fig. 8.** Performance evaluation results of the proposed method on the synthetic dataset: (a) ordinal, MSE, (b) ordinal, MSE+KLD, (c) ordinal, MSE+CCE, (d) one-hot, MSE, and (e) one-hot, MSE+KLD.

Finally, when comparing the three final embedding vector generation methods (i.e., the mean, trimmed mean, and mode vectors), we can see that the mean and trimmed mean vectors show similar good performance, while the mode vector shows relatively poor performance. This is because the mode vector reflects only some of the values in the encoding vectors and ignores the rest. On the other hand, the mean and trimmed vectors can represent the information contained in all the encoding vectors well. Especially, the trimmed mean vector sometimes shows better performance than the mean vector because it removes the effect of outlier values in the encoding vectors. Therefore, using the mean or trimmed mean vectors seems to be a more effective way to generate final embedding vectors.

Fig. 9(a) shows the precision at *k* of the proposed method, RNN autoencoder [9], and Graph2vec [11] on the synthetic dataset. For the proposed method, we used the one-hot encoding to encode node labels, MSE+KLD as the loss function, and the trimmed mean vectors to generate final embedding vectors. In Fig. 9(a), we can see that the proposed method always shows more than 96% precision at *k* regardless of *k*, while RNN autoencoder and Graph2vec show only about 50% precision at *k*. Since RNN autoencoder and Graph2vec only consider the structure of graphs and not the weights of edges in graphs, they generate almost similar embedding vectors for two different groups where graphs have similar shapes but have

very different weights (e.g., Group 1 and Group 2 in Table 1). As a result, the embedding vectors for the two different groups are hardly distinguishable from each other. Thus, for each selected graph, its *k* most similar graphs contain graphs from the two different groups in almost the same proportion (i.e., 50%:50%). On the other hand, the proposed method can generate different embedding vectors for the two different groups by considering the weights of edges in graphs.

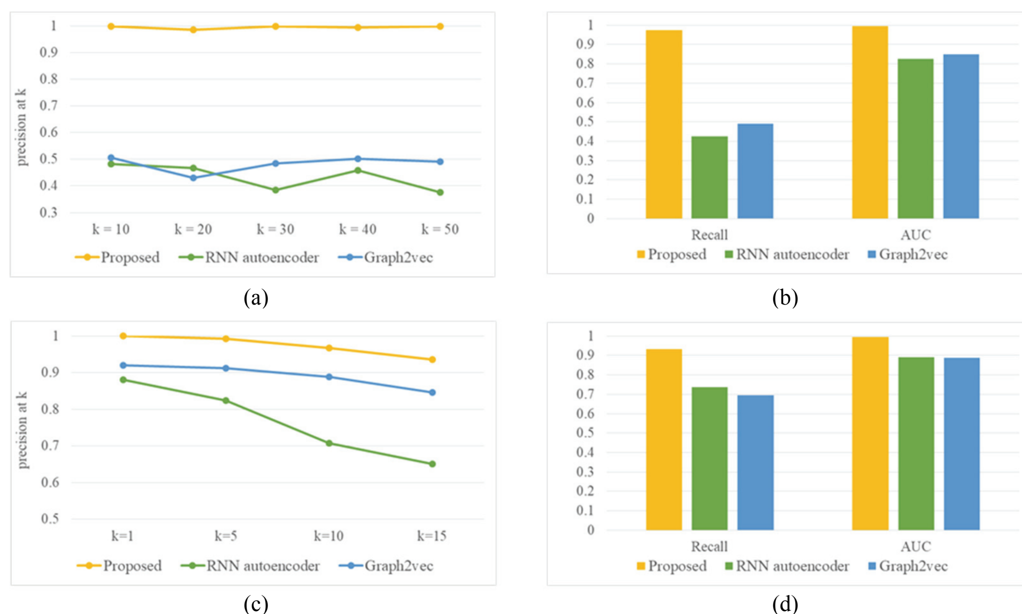


**Fig. 9.** Performance comparison of the proposed method with the existing methods: (a) synthetic dataset (precision at *k*), (b) synthetic dataset (recall and AUC), (c) real dataset (precision at *k*), and (d) real dataset (recall and AUC).

Fig. 9(b) shows the recall and area under the receiver operating characteristic curve (AUC) of the three methods on the synthetic dataset. To measure the recall, for each selected graph, we first extracted the 100 graphs with embedding vectors closest to the selected graph's embedding vector among all the graphs in the dataset. We then computed the recall as the number of graphs belonging to the same group as the selected graph in the extracted 100 graphs, divided by the number of graphs belonging to the same group as the selected graph in the whole dataset (i.e., 100). On the other hand, the AUC measures how well the ranking of graphs in terms of their distances from the selected graph reflects the true ranking. An AUC of 1 represents a perfect ranking and 0 represents a perfect inverse ranking. From Fig. 9(b), we can see that the proposed method shows better performance than the other methods also in terms of both the recall and AUC. Therefore, we can confirm that the proposed method is effective in generating embedding vectors for weighted graphs.

### 4.2.2 Performance evaluation on real dataset

In this section, we show the performance evaluation results on the real compound dataset. The real dataset consists of five groups of similar graphs, each of which consists of 20 chemical compounds. In this experiment, we first generated the embedding vectors of all the graphs in the real dataset using the proposed method, RNN autoencoder, and Graph2vec, respectively. In the case of the proposed method,

it took about 1 hour and 40 minutes to extract node-weight sequences, train an LSTM autoencoder, and generate the final embedding vectors of all the graphs. The total number of node-weight sequences extracted from all the graphs was 650. We then randomly selected five graphs from each group and, for each selected graph, extracted the $k$ graphs with embedding vectors closest to the selected graph's embedding vector among all the graphs. We measured precision at $k$ by increasing $k$ from 1 to 15. Because there are 10 types of atoms in the real dataset, we encoded each atom using the ordinal encoding as in Table 2.

**Table 2.** Atoms in the real dataset and their encoding

| Atom | C | N | O | F | P | S | Cl | As | I | Hg |
|---|---|---|---|---|---|---|---|---|---|---|
| Encode | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

Fig. 9(c) shows the precision at $k$ of the proposed method, RNN autoencoder, and Graph2vec on the real dataset. For the proposed method, we used MSE as the loss function and the trimmed mean vector as the final embedding vector generation method. In Fig. 9(c), the proposed method always shows more than 94% precision at $k$ for all $k$ values. Note that, in particular, the proposed method shows 100% precision at $k$ for $k = 1$, which corresponds to finding the most similar graph. On the other hand, we can see that RNN autoencoder and Graph2vec show only about 65%–90% precision at $k$. We can also see that the performance of RNN autoencoder decreases significantly as $k$ increases. This is because RNN autoencoder does not consider the weights of edge (i.e., the binding force between atoms) so it generates similar embedding vectors for compounds with different binding forces but similar shapes. Graph2vec also shows worse performance than the proposed method because Graph2vec only considers the degree of each node but not weights between nodes. On the other hand, the proposed method generates more effective embedding vectors for chemical compounds by considering not only the structure of compounds but also the binding force between atoms. Fig. 9(d) shows the recall and AUC of the three methods on the real dataset. Also in this case, the proposed method shows better performance than the other methods in terms of both the recall and AUC. Thus, we can conclude that embedding vectors generated by the proposed method capture similarities between weighted graphs more effectively.

# 5. Conclusion

In this paper, we propose an effective, deep learning-based graph embedding technique for weighted graphs. Unlike the previous deep learning-based graph embedding techniques, which consider only the structure of graphs and not the weights of edges in graphs, the proposed method considers both the structure and weights of graphs to generate their embedding vectors. To do so, the proposed method extracts node-weight sequences from given graphs and encodes them into fixed-length vectors using an LSTM autoencoder. Finally, the proposed method combines these encoding vectors to generate the final embedding vector for each graph.

In the proposed method, we used two encoding methods to encode node labels (i.e., the ordinal and one-hot encoding), three loss functions to train an LSTM autoencoder (i.e., mean-squared error, Kullback-Leibler divergence, and categorical cross entropy), and three generation methods to generate final embedding vectors (i.e., the mean, trimmed mean, and mode vectors). Through extensive

experiments, we investigated the performance differences of the proposed method when each of these methods is used. The experimental results on the synthetic and real datasets show that the proposed method outperforms the existing methods significantly in generating embedding vectors for weighted graphs. Therefore, we can conclude that the proposed method can be effectively used in measuring the similarity between weighted graphs.

Compared with previous work for graph embedding, this paper makes a contribution by extending the target of graph embedding to weighted graphs for the first time. Although the main idea of the proposed method itself is mainly based on [9], we extend node sequences used in [9] to node-weight sequences to consider the weights of edges in graphs. Furthermore, the contributions of this paper also come from the following: (1) In order to make the proposed method suitable for weighted graphs, there are many options or alternatives in each step, including the node encoding method, the loss function of the LSTM autoencoder, and the final embedding vector generation method. In this paper, we propose at least two or three strategies for each step, which constitute the originality of this paper. (2) Through extensive experiments, we comprehensively evaluate and analyze the effect of each of these strategies. As a result, we can gain insight about the effect of using each strategy. (3) We not only show the effectiveness and practical applicability of the proposed method by using both synthetic and real datasets, but also publish our code on GitHub. Therefore, this paper makes a practical contribution. In future work, we will study fast and effective embedding methods for large-scale weighted graphs because the execution time of the proposed method increases directly with the number of graphs as well as the number of nodes in graphs.

## Acknowledgement

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2018-0-00269, A research on safe and convenient big data processing methods).

## References

[1] M. J. J. Ghrabat, G. Ma, I. Y. Maolood, S. S. Alresheedi, and Z. A. Abduljabbar, “An effective image retrieval based on optimized genetic algorithm utilized a novel SVM-based convolutional neural network classifier,” *Human-centric Computing and Information Sciences*, vol. 9, article no. 31, 2019.

[2] D. Lee and J. H. Park, "Future trends of AI-based smart systems and services: challenges, opportunities, and solutions," *Journal of Information Processing Systems*, vol. 15, no. 4, pp. 717-723, 2019.

[3] E. Gultepe and M. Makrehchi, “Improving clustering performance using independent component analysis and unsupervised feature learning,” *Human-centric Computing and Information Sciences*, vol. 8, article no. 25, 2018.

[4] 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*, vol. 30, no. 9, pp. 1616-1637, 2018.

[5] P. Goyal and E. Ferrara, "Graph embedding techniques, applications, and performance: a survey," *Knowledge-Based Systems*, vol. 151, pp. 78-94, 2018.

[6] S. V. N. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt, "Graph kernels," *Journal of Machine Learning Research*, vol. 11, pp. 1201-1242, 2010.

[7] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61-80, 2009.

[8] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proceedings of International Conference on Learning Representations* (ICLR), Toulon, France, 2017.

[9] A. Taheri, K. Gimpel, and T. Berger-Wolf, "Learning graph representations with recurrent neural network autoencoders," in *Proceedings of the 24th ACM SIGKDD Conference of Knowledge Discovery and Data Mining: Deep Learning Day*, London, UK, 2018.

[10] P. Yanardag and S. V. N. Vishwanathan, "Deep graph kernels," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Sydney, Australia, 2015, pp. 1364-1374.

[11] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y., and S. Jaiswal, "graph2vec: learning distributed representations of graphs," in *Proceedings of 13th International Workshop on Mining and Learning with Graphs (MLG)*, Halifax, Canada, 2017.

[12] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proceedings of International Conference on Machine Learning*, Beijing, China, 2014, pp. 1188-1196.

[13] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013 [Online]. Available: https://arxiv.org/abs/1301.3781.

[14] M. Seo and K. Y. Lee, “A weighted graph embedding technique based on LSTM autoencoders,” in *Proceedings of the Korea Software Congress (KSC),* Pyeongchang, South Korea, 2019.

[15] S. Hochreiter and J. Schmidhuber. "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp.1735-1780, 1997.

[16] M. A. Kramer, "Nonlinear principal component analysis using autoassociative neural networks," *AIChE Journal*, vol. 37, no. 2, pp. 233-243, 1991.

[17] S. Maity, M. Abdel-Mottaleb, and S. S. Asfour, "Multimodal biometrics recognition from facial video with missing modalities using deep learning," *Journal of Information Processing Systems*, vol. 16, no. 1, pp. 6-29, 2020.

[18] A. Grover and J. Leskovec, "node2vec: scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, San Francisco, 2016, pp. 855-864.

[19] PubChem: an open chemistry database at the National Institutes of Health [Online]. Available: https://pubchem.ncbi.nl m.nih.gov/.

**Minji Seo** **https://orcid.org/0000-0002-4720-8250**

She received the B.S. degree from the Division of Computer Science, Sookmyung Women's University, Korea, in 2018 and M.S. degree from the Department of Computer Science, Sookmyung Women’s University, Korea, in 2020. Her research interests include databases, data mining, deep learning and graph embedding.

**Ki Yong Lee** **https://orcid.org/0000-0003-2318-671X**

He received his B.S., M.S., and Ph.D. degrees in Computer Science from KAIST, Daejeon, Republic of Korea, in 1998, 2000, and 2006, respectively. From 2006 to 2008, he worked for Samsung Electronics, Suwon, Korea as a senior engineer. From 2008 to 2010, he was a research assistant professor of the Department of Computer Science at KAIST, Daejeon, Korea. He joined the faculty of the Division of Computer Science at Sookmyung Women’s University, Seoul, in 2010, where currently he is a professor. His research interests include database systems, data mining, big data, and data streams.