LIME: Low-Cost and Incremental Learning for Dynamic Heterogeneous Information Networks

Hao Peng, Renyu Yang, *Member, IEEE*, Zheng Wang, Jianxin Li, Lifang He, *Member, IEEE*, Philip S. Yu, *Fellow, IEEE*, Albert Y. Zomaya, *Fellow, IEEE*, and Rajiv Ranjan, *Senior Member, IEEE*

Abstract—Understanding the interconnected relationships of large-scale information networks like social, scholar and Internet of Things networks is vital for tasks like recommendation and fraud detection. The vast majority of the real-world networks are inherently heterogeneous and dynamic, containing many different types of nodes and edges and can change drastically over time. The dynamicity and heterogeneity make it extremely challenging to reason about the network structure. Unfortunately, existing approaches are inadequate in modeling real-life dynamical networks as they either have strong assumption of a given stochastic process or fail to capture the heterogeneity of network structure, and they all require extensive computational resources. We introduce LIME, a better approach for modeling dynamic and heterogeneous information networks. LIME is designed to extract high-quality network representation with significantly lower memory resources and computational time over the state-of-the-arts. Unlike prior work that uses a vector to encode each network node, we exploit the semantic relationships among network nodes to encode multiple nodes with similar semantics in shared vectors. By using many fewer node vectors, our approach significantly reduces the required memory space for encoding large-scale networks. To effectively trade information sharing for reduced memory footprint, we employ the recursive neural network (RsNN) with carefully designed optimization strategies to explore the node semantics in a novel cuboid space. We then go further by showing, for the first time, how an effective incremental learning approach can be developed – with the help of RsNN, our cuboid structure, and a set of novel optimization techniques – to allow a learning framework to quickly and efficiently adapt to a constantly evolving network. We evaluate LIME by applying it to three representative network-based tasks, node classification, node clustering and anomaly detection, performing on three large-scale datasets. We compare LIME against eleven prior state-of-the-art approaches for learning network representation. Our extensive experiments demonstrate that LIME not only reduces the memory footprint by over 80% and the processing time over 2x when learning network representation but also delivers comparable performance for downstream processing tasks. We show that our incremental learning method can boost the learning time by up to 20x without compromising the quality of the learned network representation.

Index Terms—Network representation learning, heterogeneous information networks, incremental learning, memory optimization

1 INTRODUCTION

Having the ability to understand the interconnected relationships of large-scale network structures, such as social, transport, IoT and scholar networks, is crucial for many important applications like fraud and anomaly detection [1], link prediction [2], recommendation [3], etc. In a real-life setting, many networks – including social media [4], scholar networks [5], patient and drug networks [6] and IoT networks [7] – are *heterogeneous data structures*. These heterogeneous information networks (HINs) contain multiple types of objects and links, having millions or even billions of vertices [8]. The scale and complexity of real-world HINs make automated machine learning a highly attractive technology for capturing and reasoning about the relationships or semantics hidden in a large and complex structure.

The difficulty for applying machine learning to HINs, however, is that it requires the network to be represented as a set of features or *embeddings* that serve as inputs to a machine learning tool. Given that real-life HINs like social networks are unbounded, dynamically evolving graphs and that there is an infinite number of these potential features, finding the right representation for a large and evolving HIN is not trivial [3].

Efforts have been devoted to extracting useful network representations. This is now an active research field known as Network Representation Learning (NRL) [3]. The goal of NRL is to map nodes of a large-scale network to a low-dimensional embedding space. By doing so, each vertex of the network can then be represented as a low-dimensional vector of numerical values, whilst much important information of the global and local network structures can be preserved. The extracted network representations can then be used to characterize the target network and serve as input to decision models for a wide range of downstream processing tasks [9], [10].

While promising, existing approaches for NRL [11], [8],

- *Hao Peng and Jianxin Li are with Beijing Advanced Innovation Center for Big Data and Brain Computing and the State Key Laboratory of Software Development Environment, Beihang University, Beijing 100083, China. E-mail:{penghao, lijx}@act.buaa.edu.cn.*
- *Renyu Yang and Zheng Wang are with the School of Computing, University of Leeds, Leeds LS2 9JT, UK. E-mail: {r.yang1, z.wang5}@leeds.ac.uk.*
- *Lifang He is with the Department of Computer Science and Engineering, Lehigh University, Bethlehem, PA 18015 USA. E-mail: lih319@lehigh.edu.*
- *Philip S. Yu is with the Department of Computer Science, University of Illinois at Chicago, Chicago 60607, USA. E-mail: psyu@uic.edu.*
- *Albert Zomaya is with the University of Sydney, Australia. E-mail: albert.zomaya@sydney.edu.au.*
- *Rajiv Ranjan, Computing Science and Internet of Things, Newcastle University, Newcastle, UK, E-mail: raj.ranjan@newcastle.ac.uk.*

Manuscript received May 2020, revised October 2020, accepted January 2021. (Corresponding author: Jianxin Li.)

[12], [13], [14], [15], including graph-based learning methods [16], [17], are primarily concerned about static networks, assuming the network does not change over time. By ignoring the dynamicity of networks, they are inadequate in modeling many real-life networks like social networks that are constantly evolving. Some of the most recent studies try to address the dynamicity of networks: Stochastic-based approaches [18], [19], [20], [21] strongly assume the network change follows a certain stochastic process, which hardly stands in realistic networks. Temporal random walks based approaches [22], [23], [24] fail to capture the heterogeneity of network structure and thus deliver low quality NRL. Most importantly, existing approaches all require extensive memory resources and long training time to learn an effective embedding model. These drawbacks limit the practicality and the scale the technique can operate.

We present LIME[1], a better approach for learning representation for dynamic HINs. LIME is designed to learn representations for a large and dynamically changing HIN with significantly lower computational and memory overhead compared to state-of-the-art NRL techniques. To reduce the computational resource requirement, LIME maps the input HIN to a cuboid structure consisting of three directional components: rows, columns and pages, where nodes within a directional component share the same component-level embedding vector. In this way, a vertex is jointly represented by three components: a row vector, a column vector and a page vector. Since nodes in each row, column or page share the same component vector, we use only $3\sqrt[3]{n}$ vectors to represent a network of n vertices. In real terms, this means we need as few as 300 vectors to represent a graph of 1M nodes. Compared to prior work that requires one embedding vector for representing each vertex, our strategy thus significantly reduces the number of vectors and the associated computational resources for representing large networks. As the computation is performed on a much smaller number of vectors, our approach speeds up the training time considerably.

At the core of LIME is a *Recursive Neural Network* (RsNN)[2] that traverses the relationships (or edge links) of network topology. Unlike NRL methods based on the Recurrent Neural Network (RNN) [25], our approach explicitly models and exploits the different relationship types to constraint the errors of learning network embeddings. By formulating the embedding learning problem in a cuboid structure and using the RsNN to exploit the cuboid space, we can effectively trade information sharing for saving in memory and computation overhead. Our evaluation shows that our static embedding scheme has little impact on the learning performance and can even improve the performance of the downstream process task in certain scenarios.

To adapt to a dynamic network where new nodes and edges constantly emerge, LIME does not retrain the embedding network from scratch on the entire network because doing so would be too slow and resource expensive. Instead, it incrementally calibrates the learned network embeddings based on the changed network structures. One of the key challenges for our incremental learning strategy is how to minimize the error of the objective function as we only perform computation on a subset of nodes and relationships of a vast network. To this end, we first formulate the objective function for incremental learning, so that we can apply the widely used stochastic gradient descent (SGD) training method to effectively calibrate and minimize the error of incremental learning of local nodes. We then go further by showing how the global, network-wide optimization problem can be translated into a standard minimum weight perfect matching problem [26]. This enables us to apply the well-established dynamic minimum cost maximum flow (MCMF) algorithm to further optimize the loss function globally across a large network. We note that LIME is the first attempt in applying incremental learning and the dynamic MCMF algorithm to address NRL for dynamic HINs.

We demonstrate the benefits of LIME by applying it to three large-scale datasets, including two scholar networks and a cyber-physical network. We evaluate LIME by using it to support three representative downstream tasks, node classification, node clustering and anomaly detection, and compare it against eleven state-of-the-art NRL techniques [13], [11], [12], [8], [27], [28], [18], [20], [23], [22], [29]. Experimental results show that LIME is highly effective and resource-efficient in learning representations for both static and dynamic HINs. Compared to prior NRL techniques, LIME reduces the memory footprint by 4 to 6 times. It cuts down the static and incremental learning time by up to 2 and 20 times, respectively. We show that such great advantages in the memory usage and computation time do not come at the cost of lower performance for the downstream processing task. Instead, in certain scenarios, LIME can improve the performance of the subsequent task by up to 3% over the best-performing NRL baseline. As a result, LIME represents a new way of learning network representations, which exhibits a better scalability with less memory and computational cost over existing NRL techniques.

This paper makes the following contributions:

- It proposes a novel resource-efficient NRL model based on RsNN for learning static embeddings of HINs (x 4).
- It presents the first incremental embedding scheme for dynamic HINs based on changed network structures (x 5). LIME advances prior works by showing how efficient incremental learning can be achieved by formulating the optimization space and by applying the dynamic MCMF algorithm.
- It demonstrates how static and incremental NRL techniques can be combined together to effectively support a wide range of network-related tasks (x 7).

LIME is open-sourced and can be downloaded from `https://github.com/RingBDStack/LIME`.

2 BACKGROUND

In this section, we introduce HINs and NRL and formulate the scope of this work.

1. LIME =Low-cost and Incremental network eMbedding Engine.

2. Not to be confused with a Recurrent Neural Network (RNN) (rb.gy/m93yqi). A Recursive Neural Network (RsNN) (rb.gy/picdx4) is a hierarchical network where the input is processed hierarchically in a tree fashion. This is different from an RNN where the network unfolds over time, which is used for sequential inputs where the time factor is the main differentiating factor between the elements of the sequence.

2.1 Information Networks

LIME is a general framework for learning network representations. In this work, we target HINs that have heterogeneous structures and are dynamically evolving.

Example information networks include scholar networks like DBLP, social networks like Twitter, and cyber-physical systems and Internet of Things (IoT) networks. Real-life information networks have different structures consisting of multi-typed entities and relationships. Herein, a relationship refers to the link between entities in a network system such as the interconnection between an IoT sensor and an edge device, interactions between social network users, collaborations among co-authors in a scholar community, etc. For example, the Twitter network contains multiple entities (or node types) like users, tweets, hashtags and terms, etc., as well as relationships encompassing follows among users, posts between users and tweets, replies between tweets, etc.

Many information networks are dynamic structures, because new entities and relationships can be added to the network over time. A typical real-life network can have millions or billions of entities. For example, there are over 2.2M authors and over 5M papers in DBLP. The interconnections of entities in DBLP, in turn, lead to hundreds of millions of relationships among entities. The massive scale and the dynamically changing behavior of an information network make it extremely challenging to extract and ascertain the subtle interconnected relationships of the network.

2.2 Network Representation Learning

To extract knowledge from a large network, we need to find ways to capture the essential characteristics and structures of the network. This is typically done by representing the network using a fixed-length vector or matrix of numerical values. The idea is to encode the network vertices using latent, low-dimensional representations (or embeddings), which can highly summarize informative characteristics of the network and preserve information like the network topology structure, node content, and other neighborhood information. Due to the high computational complexity of traditional network representation learning, a popular method is to use random walk method to obtain an approximation of the network structure[3]. This line of research is known as network representation learning (NRL).

After the new node embeddings are learned, network analytic tasks (such as graph visualization, node classification and clustering, and link prediction, etc.) can then be performed by applying vector-based machine learning algorithms to the new representation space. NRL allows us to apply many well-established machine learning algorithms to a large, complex graph structure. Our work develops a new approach to learn network representations for dynamic, heterogeneous information networks, aiming to provide better scalability with lower memory and computational cost for NRL.

2.3 Preliminaries

In this work, we follow the terminologies used in the seminar work of [13], [31] to define a dynamic HIN and the network embedding.

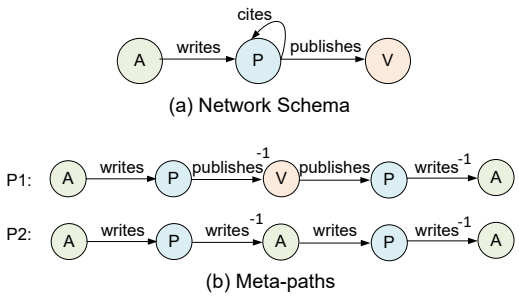

Fig. 1. A simple scholar schema (a) and two possible meta-paths (b) of a network under this schema. Diagrams are reproduced from [30].

2.3.1 Dynamic heterogeneous information networks

A dynamic HIN is a temporal graph $G(t) = (V; E; A; R)$ with an entity type mapping $: V \to A$ and a relationship type mapping $: E \to R$, where V and E represent entity set and link set, respectively, while R and A denote the type set of corresponding entities and links. Here, t denotes the current time-stamp. As illustrated in Fig. 1a, at a given moment t, a scholar network (e.g., DBLP), G, consists of three node types: *Author*, *Paper* and *Venue*, and three types of links (or relationships): "*an author writes a paper*", "*a paper cites another paper*", and "*a paper is published in a venue*".

In real world settings, there may exist multiple types of entities and relationships, i.e., $|A| > 1$ and $|R| > 1$. Most notably, each entity and link are annotated by chronological addition. For example, at time-stamp $t + 1$, the network can be expressed as $G(t + 1) = (V + \Delta V; E + \Delta E; A; R)$, where ΔV and ΔE represent the recently-added entities and links, respectively. Our current implementation assumes that the types of a given entity and its relationships do not change, but new entities and relationships can be added into the network. For example, in a scholar network, an authoring entity and its existing relationships, such as "*author X writes paper Y*", would rarely change, but new authors or newly published papers can be added into the network over time.

It is worth noting that the abstract concepts of an entity and a link are literally instantiated by a *node* and an *edge* respectively in a HIN instance. To aid clarity, we use *node* and *edge* to represent the HIN objects in the rest of the paper.

2.3.2 Dynamic meta-path guided random walks

Since our neural network (i.e., RsNN) works on a sequential sequence, we need to translate the graph structure of a network to a linearized sequence. We achieve this by applying meta-path-based random walks to the target network. This technique is proven to be effective in prior HIN embedding studies [13], [32].

In the context of NRL, a meta-path is a path that connects a pair of network nodes. It describes the semantic relationship between nodes, and the mining of this semantic relationship is the cornerstone of subsequent tasks. Take a simple scholar network shown in Fig. 1a as an example. This network consists of three types of nodes, authors (A), papers (P), and venues (V), and three types of edges, "*author X writes paper Y*", "*paper X cites paper Y*", and "*paper X published in venue Y*". Fig. 1b shows two possible meta-paths, where P1: "A-P-V-P-A" represents papers published by two authors in the same venue, and P2: "A-P-A-P-A"

describes that two authors share the same co-author. Here, write ^{1} and publish ^{1} represent a backward edge on the network.

Since we are dealing with a very large graph, it is prohibitively expensive to exhaustively enumerate all meta-paths of a network. Instead, existing techniques [13], [32] typically perform meta-path-guided random walks to sample node pairs, hoping that a carefully designed random sampling scheme would capture much of the semantic and structural correlations between different node pairs. In this work, we apply meta-path-guided random walks to dynamic HINs.

Given dynamic HINs $G(t)$ and $G(t+1)$, a dynamic meta-paths-guided random walk aims to continuously generate sequences of heterogeneous nodes (paths of multiple types of nodes) to feed recursive neural network, that contain the newly added nodes $v \in V$ and edges $e \in E$, guided by meaningful meta-paths. We first assume that the length of node sequences generated by the dynamic meta-path guided random walk at t time-stamp is T, and the length of node sequences generated at $t+1$ time-stamp is T. We guarantee that $T + T$ is equivalent to the length of node sequences generated by cold start of meta-paths guided random walk on the HIN $G(t+1)$. Moreover, we assume that the probability of a certain meta-path m during the walking is $_m$ and ensure that the same probability is used when walking in both $G(t)$ and $G(t+1)$ in order to obtain an unbiased meta-paths-guided random walk. In practice, we set an equal probability $_m$ for each meta-path to avoid introducing any bias among meta-paths. In this paper, we use dynamic meta-paths-guided random walks to assist the incremental learning described in x 5.

3 System Overview

LIME performs NRL using a two-step approach. It first learns the node embeddings of a static network. It then applies incremental learning to update the node embeddings for a changed network.

3.1 Cuboid abstraction structure

A key innovation of our approach is how we encode the node information. In this work, we map the input network nodes into a cuboid space with three directional components, rows, columns and pages Fig. 4 gives an example cuboid of a scholar network. Using our approach, nodes within the same directional component will share the same embedding vector assigned to that component. For example, nodes in the i-th row will share the same row embedding vector, x_i^r. Similarly, nodes in the j-th column and k-th page will share the same column and page embedding vectors, x_j^c and x_k^p respectively. As a result, a network node in the i-th row, j-th column and k-th page will be jointly represented by three components, $(x_i^r; x_j^c; x_k^p)$. We call this a 3-component (3-C) node representation. By sharing the embedding vector among nodes in the same directional component for a network with n nodes, we need only $3\sqrt[3]{n}$ unique vectors for encoding the node embeddings. Since existing NRL methods [3] require at least n unique vectors for n network nodes, our cuboid representation thus significantly reduces the number of vectors and the memory storage space for node embeddings. Note that we use the same cuboid structure to encode the output node representation given by our learning framework.

Fig. 2. System Architecture of the LIME Engine

3.2 Learning frameworks

The proposed learning framework for static network embeddings, namely HETERRSNN, is based on the RsNN because RsNN can better model complex hierarchical structures over the RNN alternative. However, native RsNN is a supervised model and merely operates on homogeneous network. We also do not use the graph neural network (GNN) as it requires significantly more memory resource for representing the graph structure and for learning graph embeddings, which hence does not scale well to large networks [3]. HETERRSNN exploits the cuboid representation to reduce the memory and computation overhead for learning node representation. Its goal is to map all HIN nodes into a 3-dimensional cuboid while maintaining the pertaining attributes and relationships. This is achieved by using a carefully designed objective function to maximize the network probability in predicting the right node of a network path, by considering the multiple node and edge types. Like other mainstream heterogeneous network embedding models [13], [32], we employ meta-path-guided random walks (see also x 2.3.2) to capture both semantic and structural relationship among different nodes and construct heterogeneous neighborhood for each node. We describe HETERRSNN in more details in x 4.

To adapt to changing network structures, we extend HETERRSNN with the capability to perform incremental learning, for which we refer to as HETERRSNN++. This network is designed to update the node representation obtained for the previous network observed at a previous time epoch t (i.e., a month before), by taking into consideration the changed network structures (i.e., new nodes and edges) observed at the current update epoch $t+1$. A notable novel aspect of HETERRSNN++ is that it employs the dynamic minimum cost maximum flow (MCMF) algorithm to adjust the node mapping in the cuboid space to minimize the error of incremental learning. HETERRSNN++ is the first work in performing incremental NRL for HINs. We describe this network in more details in x 5.

Our work can be applied to arbitrary HINs, be it a dense or sparse graph. In this paper, we focus on tackling the heterogeneity of nodes and edges of a large-scale HIN. We implement LIME as a service platform. Fig. 2 depicts

Fig. 3. Applying HETERRSNN to the P1 meta-path ("A-P-V-P-A") shown in Fig. 1b. Here x^r, x^c, x^p and a denote the row vector, column vector, page vector and a node type of the target network. Unlike an RNN where weights are passed as a linear pipeline (e.g., the V edge from $h^r_{i\ 1}$ to $h^c_{i\ 1}$), HETERRSNN enables information propagation in a hierarchical tree fashion (e.g., the V edge from $h^r_{i\ 1}$ to $h^p_{i\ 1}$).

the architecture of the LIME engine and the underpinning modules. A request for HIN embedding is handled and dispatched by the LIME Engine as an embedding task. The processed results will be aggregated and sent back to the user. We execute the HETERRSNN and HETERRSNN++ as instances in Docker Container. Job dispatch and resource management are also coordinated by the LIME Engine.

4 LEARNING STATIC NETWORK EMBEDDINGS

Existing embedding learning approaches designed for memory-efficient [25] typically use an RNN for embedding learning. As a significant departure from prior work, HETERRSNN employs the RsNN and maps all nodes in a HIN into a cuboid space to learn node embeddings. In x 7.2 we show that our RsNN-based approach significantly outperforms the RNN alternative.

4.1 Overview of HETERRSNN

With HETERRSNN, NRL is performed in a cuboid space described in x 3.1. We initialize the shared row, column and page vectors with random values, and network nodes are first randomly assigned to the cuboid space. At each training epoch, we update the shared embedding vector for each directional component, and the node location in the cuboid. To update the embedding vector, we fix the node location in the cuboid (x 4.2). Then, we re-adjust the node locations in the cuboid based on the learned embedding vectors (x 4.3). When training terminates, HETERRSNN outputs the node embeddings (in a cuboid structure) as the representation of the target network.

Working example. We use the scholar network given in Fig. 1 as a working example to explain how HETERRSNN learns the component vectors (i.e., the network representation) as depicted in Fig. 3. Here, HETERRSNN takes as input a node vector (x 3.1) of e.g., an author (A) or a paper (P) node in the P1 meta-path of Fig. 1b, "A-P-V-P-A". It then uses an encoder-decoder scheme to predict the probability of each paper node to be the next node when seeing A, or the probability of each venue to be the next node when seeing P of the meta-path P1. In other words, for the meta-path, P1, HETERRSNN predicts, given an author node A, which paper this author is likely to be a co-author; similarly, it also predicts, given an input relationship "author A writes paper P", which venue, V, is likely to be the publication venue of paper P. In essence, HETERRSNN traverses a given meta-path to find node representation that can maximize the probability to predict the right node given a partially seen network path. We note that the same learning strategy applies to any other meta-paths of the same network or other types of networks. As outlined in x 2.3.2, the node sequences are generated by applying dynamic meta-path-guided random walks to the target network.

4.2 Learning Component Embeddings

4.2.1 Training objective

Our training goal is to maximize the likelihood of correctly predicting the next node in a meta-path, given a partially seen meta path. In other words, we want to minimize the negative log-likelihood of the next node in the node sequence. This is equivalent to optimizing the cross-entropy between the target probability distribution and the predicted one given by HeterRsNN The probability is determined by its row probability P_r, column probability P_c and page probability with given node type P_{pja}. Hence, the overall negative log-likelihood (NLL) can be formalized as follows:

$$NLL = \sum_{t=1}^{T} \log P_r(N_t) \quad \log P_c(N_t) \quad \log P_{pja}(N_t); \quad (1)$$

where T is the length of node sequences, and N_t refers to t-th node in the node sequences.

The negative log-likelihood can be also expanded with respect to nodes, i.e., $NLL = \sum_{v=1}^{|V|} NNL_v$, where $|V|$ refers to the total number of unique nodes in HIN, and NLL_v is the negative log-likelihood for a node v in the HIN. Meanwhile, from the perspective of our cuboid structure, NLL_v is equal to $l(v; r_{(v)}; c_{(v)}; p_{(v)})$, where $(r_{(v)}; c_{(v)}; p_{(v)})$ is the position of the node v in the cuboid. In this context, the negative log-likelihood for a node v can be expressed as:

$$\begin{aligned} NLL_v &= \sum_{pos \in S_v} \log P_r(N_{pos}) \quad \log P_c(N_{pos}) \quad \log P_{pja}(N_{pos}) \\ &= l_r(v; r_{(v)}) + l_c(v; c_{(v)}) + l_p(v; p_{(v)}); \end{aligned} \quad (2)$$

where l_r, l_c and l_p refer to the row loss, column loss and page loss, and S_v is the position set of node v in the node sequences.

4.2.2 Embedding learning

Using the working example for predicting P given an input author node A, the problem of learning node representation for a paper node P_i can be formulated as follows. Let n be the dimension of a row, column or page input vector, and m be the dimension of a hidden state vector of HETERSNN. We use a to denote the type of a network node, whether it is an author, paper, or venue in our working example (Fig. 1). For the input node A_{i-1} of meta-path P1, "A-P-V-P-A", we wish to exploit the column vector $x_{i-1}^c \in R^n$, page vector $x_{i-1}^p \in R^n$, and row vector $x_i^r \in R^n$, hidden state vectors $h_{i-1}^c \in R^m; h_{i-1}^p \in R^m$, and $h_i^r \in R^m$ as well as the state value of node type a to estimate the probability of node P_i (i.e., the second node of the "A-P-V-P-A" meta-path). In reality, the column, row and page vectors are derived from input-embedding matrices X^c, X^r and $X^p \in R^{n \times \sqrt[3]{|V|}}$, respectively. As a result (see Fig. 3), the three hidden state vectors $h_{i-1}^c; h_{i-1}^p; h_i^r$ can be produced by applying the following recursive operations:

$$\begin{aligned} h_{i-1}^c &= \sigma(W x_{i-1}^c + V h_{i-1}^r + b) \\ h_{i-1}^p &= \sigma(W x_{i-1}^p + V h_{i-1}^r + b) \\ h_i^r &= \sigma(W x_i^r + U(h_{i-1}^c + h_{i-1}^p) + b); \end{aligned} \tag{3}$$

where $W \in R^{m \times n}; U \in R^{m \times m}; V \in R^{m \times m}; b \in R^m$ are parameters of affine transformations, and σ is a non-linear activation sigmoid function. Obviously, by using different parameters V and U and their combinations, the above operations form a recursive neural network, not the traditional recurrent neural network.

Meanwhile, the probability $P(N_i)$ of node N_i is determined by its row probability $P_r(N_i)$, column probability $P_c(N_i)$ and page conditional probability $P_{pja}(N_i)$ with the same type a as P_i:

$$\begin{aligned} P_r(N_i) &= \frac{\exp(h_{i-1}^c \cdot y_{r(N_i)}^r)}{\sum_{i \in S_r} \exp(h_{i-1}^c \cdot y_i^r)} \\ P_c(N_i) &= \frac{\exp(h_{i-1}^p \cdot y_{c(N_i)}^c)}{\sum_{i \in S_c} \exp(h_{i-1}^p \cdot y_i^c)} \\ P_{pja}(N_i) &= \frac{\exp(h_i^r \cdot y_{p(N_i)}^p)}{\sum_{i \in S_{pja}} \exp(h_i^r \cdot y_i^p)}; \end{aligned} \tag{4}$$

$$P(P_i) = P_r(N_i) \cdot P_c(N_i) \cdot P_{pja}(N_i) \tag{5}$$

where $r(N_i)$, $c(N_i)$ and $p(N_i)$ are the row index, column index, and page index of N_i, respectively. $y_i^r \in R^{m \times \sqrt[3]{|V|}}$ is the i-th vector of $Y^r \in R^{m \times \sqrt[3]{|V|}}$ while $y_i^c \in R^m$ is the i-th vector of $Y^c \in R^{m \times \sqrt[3]{|V|}}$ and $y_i^p \in R^m$ is the i-th vector of $Y^p \in R^{m \times \sqrt[3]{|V|}}$. S_r, S_c, and S_p denote the set of rows, columns, and pages of node cuboid, respectively. We note that a similar learning process is applied to estimating the probability $P(V_{i+1})$ of node V_{i+1} when seeing P_i as the model input in Fig. 3.

Unlike a RNN, HETERSNN enables information propagation in a hierarchical tree fashion (e.g., the V edge from h_{i-1}^r to h_{i-1}^p and the U edge from h_{i-1}^c to h_i^r) of Fig. 3. This allows us to better learn and aggregate the semantical and structural information among the three components of our cuboid structure.

As shown in Fig. 3, given the input column vector x_{i-1}^c and page vector x_{i-1}^p of the $i-1$-th node, we first infer the row probability $P_r(N_i)$ and the column probability $P_c(N_i)$ of the i-th node. Next, we choose the indexes of the row and column with the largest probabilities of $P_r(N_i)$ and $P_c(N_i)$ to look up the next input row vector x_i^r and heterogeneous type a. We can therefore infer the largest conditional page probability $P_{pja}(N_i)$ of the i-th node. Therefore, the computational complexity of this forward training can be proved $O(\sqrt[3]{|V|}T)$, where T is the length of the generated node sequences.

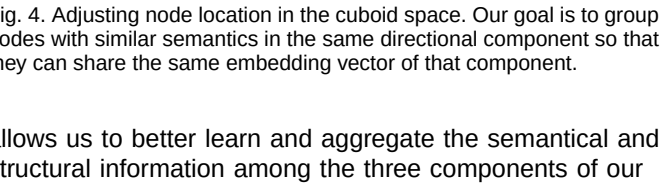

Fig. 4. Adjusting node location in the cuboid space. Our goal is to group nodes with similar semantics in the same directional component so that they can share the same embedding vector of that component.

4.3 Node Placement Optimization

Since nodes in the same directional component, be it a row, column or page, share a single component-level vector, we can further optimize the loss function by adjusting the placement of nodes in the cuboid. Our intention is to group nodes with similar semantics, e.g., authors who tend to publish in the same venue, in the same component. Doing so can reduce the information lost when trading the number of node vectors for memory footprint.

As an example, consider the cuboid shown in Fig. 4 for a scholar network. During training, we move publication-venue node HPCA from its initial position at the j-th column to sit at the cross-section defined by the column that is shared by other systems-related venues like TC, TPDS, SOSP, and FAST. Likewise, node SDM can be moved from its initial position at the j-th column to sit together with data-mining venues like TKDE, KDD, ICDM, and WSDM. By so doing, we group nodes that are likely to share similar semantics together so that we can use a single shared component vector across all nodes in a directional component.

4.3.1 Training objective for node placement

Assume we want to move node v from its initially assigned location $(r(v); c(v); p(v))$ to a new location, $(i; j; k)$, in the

cuboid. We can independently calculate (i.e., by fixing the other two directions when moving node v in one direction, and keeping other nodes unchanged) the loss for the row $l_r(v;i)$, column $l_c(v;j)$ and page $l_p(v;k)$. According to the loss function given in Eq. 2, the total loss, $l(v;i;j;k)$, of the new location, $(i;j;k)$, for node v, due to this movement is $l_r(v;i) + l_c(v;j) + l_p(v;k)$. Here, each term of the total loss would have already been computed when learning the component embeddings. This is because to predict the next node we need to compute the probability (i.e., in Eq. 4, $h_{i\ 1}^{r;c;p}\ y_{r(P_i)}$) of all the nodes in the node sequences.

As a result, $l_r(v;i)$ is the sum of $\log(\frac{\exp(h_{t\ 1}^{c}\ y_{r(n)}^{r})}{\sum_{i2S_r}\exp(h_{t\ 1}^{c}\ y_{i}^{r})})$ over all the occurrences of node v in the node sequences seen during learning component embeddings; $l_c(v;i)$ and $l_p(v;i)$ are computed in the same way. Hence, after we calculate $l(v;i;j;k)$ for all possible new locations, $(i;j;k)$, we can translate the node reallocation into the following optimization problem:

$$\begin{aligned} \min_g \quad & \sum_{(v;i;j;k)} l(v;i;j;k)\,g(v;i;j;k) \quad \text{subject to} \\ & \sum_{(i;j;k)} g(v;i;j;k) = 1 \quad \forall v \in V; \\ & \sum_{v} g(v;i;j;k) = 1 \quad \forall i \in S_r; j \in S_c; k \in S_p; \\ & g(v;i;j;k) \in \{0;1\}, \quad \forall v \in V; i \in S_r; j \in S_c; k \in S_p; \end{aligned} \tag{6}$$

where $g(v;i;j;k) = 1$ indicates node v is assigned to $(i;j;k)$ in the cuboid, and S_r, S_c and S_p denote the set of nodes in the row, column and page directions, respectively.

4.3.2 Solving node placement optimization

Inspired by [25], we convert the above optimization problem to the standard minimum weight perfect matching (MWPM) problem [26]. This is done by defining a weighted bipartite graph [3] $BG = (V;E)$ with $V = (V;S_r \cup S_c \cup S_p)$, in which the weight of the edge in connecting a node $v \in V$ and location/node $(i;j;k) \in S_r \cup S_c \cup S_p$ is the loss $l(v;i;j;k)$ of node v. Specifically, we intend to find a set of edges so that all vertices in graph BG are matched, and the sum of the edge weights (and hence the loss) of the edge subset can be as small as possible. Here, a *matching* is a set of edges, no two sharing a vertex and a matching is perfect if all vertices are matched.

The MWPM problem has been extensively studied in the literature, and one of the widely used solutions is the minimum cost maximum flow (MCMF) algorithm. However, the computational complexity of MCMF is $O(|V|^3)$, which would still be expensive for a large network. To reduce the computational overhead, we leverage linear-time approximation [33], [34], with respect to the edge number of the bipartite graph, i.e., $|E| = |V|^2$, to find a nearly-good solution. To this end, we employ an Improved Path Growing Algorithm (IPGA') [34] to solve the node reallocation. We choose IPGA' because it is shown to be more computationally efficient and more accurate [34], [35] than the other alternative used in prior embeddings learning work tuned for resource usage [25].

3. A graph is bipartite if its vertices can be colored with two colors such that each edge has ends (or vertices) of different colors.

4.4 Time Complexity of HeterRSNN

The time complexity of training HeterRSNN comes from two parts, learning on the component embeddings and performing node reallocation using IPGA'. The complexity of the former and the latter are $O(\sqrt{|V|}T)$ (see Sec. 4.2.2) and $O(|V|^2)$ respectively, where $|V|$ is the total number of nodes in the HIN, and T is the total length of node sequences. Putting together, the overall time complexity of HeterRSNN model is $O((\sqrt{|V|}T + |V|^2)K)$, where K is the number of training epochs. This is determined by the larger one between $\sqrt{|V|}T$ and $|V|^2$.

5 Incremental Learning

5.1 Modeling Network Changes

We periodically update the existing node embeddings by considering the new nodes and edges added into the target network. This is done by first applying dynamic meta-paths-guided random walks (Sec. 2.3.2) to break down the new nodes and edges seen at time $t+1$ with respect to the network observed at the previous timestamp t. Specifically, we formulate the new node set V' and the new length of node sequences T' at time $t+1$ using increments ($\ $)[4], $V' = V + \ V$ and $T' = T + \ T$ with respect to the ones seen at time t.

Furthermore, due to the introduction of new network nodes, we also need to update the cuboid structure used for learning node embeddings. We decompose the overall loss function for learning representation for the changed network, $G(t+1)$, as:

$$NLL' = \sum_{t=1}^{T} \log P(N_t) + \sum_{t=T+1}^{T+\ T} \log P(N_t); \tag{7}$$

where N_t refers to tth node in generated node sequences. Here, we leverage the learned parameters of HeterRSNN and component embedding vectors obtained from $G(t)$ to compute the first term, $\sum_{t=1}^{T} \log P(N_t)$, of the loss function. We describe this strategy in the next subsection. Like learning static embeddings, at each training iteration, we perform two optimizations: learning the component embeddings (Sec. 5.2) and adjust the node placement in the cuboid (Sec. 5.3).

5.2 Design of HeterRSNN++

To reduce the training overhead for NRL for a changed network, HeterRSNN++ leverages the parameters and node vectors of HeterRSNN that was trained on an early version of the target network to update the node embeddings in each directional component (i.e., row, column and page). We achieve this using a two-step approach described as follows.

Step-1: Inheriting parameters and vectors. We inherit all the parameters and vectors for the target network, $G(t)$, observed at the previous timestamp, t, which have been trained in using the loss function defined in Eq. 2. More

4. V and T can be generally referred to as adding or removing network nodes and the node sequences corresponding to those nodes. Namely, the negative delta indicates the removal of the node's contribution to the loss function.

speci cally, the parameters and vectors to be inherited include $W \in \mathbb{R}^{m \times n}$; $U \in \mathbb{R}^{m \times m}$; $V \in \mathbb{R}^{m \times m}$; $b \in \mathbb{R}^{m}$; $X^r \in \mathbb{R}^{n \times \sqrt{jVj}}$; $X^c \in \mathbb{R}^{n \times \sqrt{jVj}}$; $X^p \in \mathbb{R}^{n \times \sqrt{jVj}}$; $Y^c \in \mathbb{R}^{m \times \sqrt{jVj}}$, $Y^r \in \mathbb{R}^{m \times \sqrt{jVj}}$ and $Y^p \in \mathbb{R}^{m \times \sqrt{jVj}}$. We use these parameters to initialize the newly-added vectors with the same settings in the function Eq. 2.

Step-2: Calibration. Simply using the inherited parameters and embedding vectors for the rst term of Eq. 7 will inevitably introduce large errors. To minimize the loss function de ned in Eq. 7, we calculate the difference in the loss, NLL', due to accumulated computation errors. We calibrate and reduce NLL' using a standard SGD method. To do so, we also breakdown NLL' into three directional components, corresponding to rows, columns and pages in our cuboid structure:

$$NLL' = \sum_{t=1}^{|T|} (\log \sum_{i \in S'_r} \exp(h^c_{t-1} y^r_i) + \log \sum_{i \in S'_c} \exp(h^p_{t-1} y^c_i) + \log \sum_{i \in S'_p} \exp(h^r_t y^p_i)) - |T| \log RCP, \quad (8)$$

where

$$R = \sum_{t=1}^{|T|} \sum_{i \in S_r} \exp(h^c_{t-1} y^r_i); \quad (9)$$

$$C = \sum_{t=1}^{|T|} \sum_{i \in S_c} \exp(h^p_{t-1} y^c_i); \quad (10)$$

$$P = \sum_{t=1}^{|T|} \sum_{i \in S_p} \exp(h^r_t y^p_i). \quad (11)$$

The breakdown allows us to update some of the parameters and vectors, i.e., $y^c_i \in S'_c$, $y^r_i \in S'_r$, and $y^p_i \in S'_p$, using SGD, while keeping others inherited vectors and af ning transformation parameters unchanged. By only learning and updating a subset of the inherited parameters and vectors, we can accelerate the process for learning and update component embedding vectors. We can therefore merely update partial parameters $y^c_i \in S'_c$, $y^r_i \in S'_r$, and $y^p_i \in S'_p$ vectors with SGD in Eq. 8 whilst xing the others inherited vectors and af ning transformations parameters to accelerate the procedure above.

Beside from the inherent errors introduced from the parameter inheritance, we also need to consider the incremental portion (i.e., changed network structures) in the second term of Eq. 7. To this end, we use the newly generated corpus T to train and update all parameters and vectors again with SGD.

Fault tolerance. Since the network is dynamically evolved, fault tolerance should not be ignored. Our incremental learning scheme offers a way to rapid recovery from the backward embedding vectors. Accordingly, different portions of the loss function 7 can be rebuilt if parts of the network data get lost.

5.3 Node Placement for Dynamic Networks

We now describe how to adjust the node placement in the cuboid during incremental learning, like what we do for learning static embeddings (x 4.3).

As discussed in x 4.3.2, we translate the problem of node placement optimization into an MWPM problem builds around a weighted bipartite graph, for which we solve by using an MCMF algorithm with linear-time approximation. Algorithms for solving MWPM within a bipartite graph are based on the idea of augmenting paths, de ned as follows. In graph theory, a matching is a set of edges, no two sharing a vertex (see also x 4.3.2), and an alternating path is a path whose edges are alternately in and out of the matching. An augmenting path is an alternating path that starts from and ends on unmatched vertices.

Existing linear-time approximation methods, like LAM [33] used in [25] and IPGA' [34], [35] used for static embeddings in this work, cannot solve the MWPM problem for a dynamic bipartite graph. Alternative MCMF algorithms, like the zkw algorithm [36] and the Kuhn-Munkres algorithm, are not applicable to our problem either, because they cannot effectively limit the range of nding augmenting paths when new nodes are added to the graph. We also cannot use the Dijkstra's shortest path algorithm to nd the augmenting path in our weighted bipartite graph, because it cannot handle edge with a negative weight.

To address the above limitations, we employ a Edmonds-Karp algorithm based the shortest path faster algorithm (SPFA) [37], namely EK-SPFA. This algorithm is a good t for our problem as it can effectively nd augmenting paths in a dynamic, weighted directed bipartite graph with negative-weight edges. With EK-SPFA in place, we then apply the optimization strategy as in x 4.3 for node placement optimization, but this time we use a new objective function for incremental learning (Eq. 7). Incremental learning is achieved by breaking down the changes in the node sets in each row, column and page of the cuboid, i.e., $S'_r = S_r + \Delta S_r$, $S'_c = S_c + \Delta S_c$, and $S'_p = S_p + \Delta S_p$, where ΔS_r, ΔS_c and ΔS_p respectively denote the set of nodes in the row, column and page component. We use these to compute T in the second term of the loss function given in Eq. 7.

5.4 Time Complexity of HIERSNN++

The worst-case running time of SPFA is $O(|V||E|)$, where $|E|$ is number of bidirectional edges. However, experiments suggest the average running time of SPFA is $O(k|E|)$, where k is the number of times each node entering a queue and generally meets $k \leq 2$. Since the step for nding augmenting paths runs less than $|V|$ times, and each step theoretically takes $O(|V||E|)$ with an average of $O(k|E|)$, the time complexity of our node placement algorithm for incremental learning is $O(|V|^2|E|)$ in theory and $O(k|V||E|)$ on average in practice. Here, E equals $2|V|^2 + 4|V|$ in our problem. Putting together, for our solution, the time complexity is bounded to $O(|V|^4)$ in theory and $O(k|V|^3)$ on average. It is worth noting that $|V|$ is same with the scale of the vertices of the entire dynamic bipartite graph BG, rather than the scale of the incremental vertices $\Delta|V|$ in the bipartite graph. Since the EK-SPFA algorithm is naturally compatible with dynamic network evolution, a smaller increment can help to signi cantly reduce the time complexity.

6 Experimental Setup

6.1 Hardware and Software

We evaluate the LIME Engine using a 16-node GPU cluster, where each node consists of a 64-core Intel Xeon CPU E5-2680 v4@2.40GHz with 512GB RAM and four NVIDIA Tesla P100 GPUs. The server nodes run Ubuntu 20.04 LTS with Linux kernel v.5.4.0. We implement LIME using Tensorflow v1.4.0. In the experiment, we dispatch each network embedding tasks to run on a lot of server nodes.

6.2 Model Training

For all embedding methods, we use the same set of hyper-parameters, including the learning rate (0.01), mini-batch size (64), and the multi-threading number (32) and the number of negative samples per positive (5). For random walks (x 2.3.2), we set the number of walks per node, the max walk length to form a node sequence, and the neighborhood size to be 1,000, 100, 7, respectively. As a result, a mini-batch size of 64 represents a batched training on a sequence of 64 1,000 nodes.

6.3 Datasets

We use three heterogeneous networks, including the DBLP dataset[5], Aminer Computer Sciences (CS) dataset [38] and an cyber-physical datastream, described as follows.

DBLP. This dataset consists of over 6.5 million nodes of four types: 2,251,371 authors (A), 2,476 organizations (O), and 4,314,846 papers (P) from 5,744 publication venues (V). There are three types of relationships: organization affiliates authors (O–A), authors writes papers (A–P), and papers published in venues (P-V). This dataset contains data up to 2018. We construct the heterogeneous information network following [13] and use 4 types of meta-paths. We assign each month's new nodes and edges to a new time epoch based on combinations of a publication venue, papers, new authors and new organizations.

AMiner. This scholar network dataset consists of over 4 million nodes of three types: 1,693,531 authors (A) and 3,194,405 papers (P) from 3,883 publication venues (V) held until 2016. We use AMiner heterogeneous collaboration network constructed by [13] and use 3 types of meta-paths. This dataset contains two types of relations: authors writes papers (A-P), and papers published in venues (P-V). We follow [13] to match the eight research fields (or categories[6]) for publication venues grouped by Google Scholar [7] with those in AMiner dataset to get labeled venues. We assume that a specific researcher belongs to a particular area if the researcher has over ten papers were published in corresponding venues of that area.

CTI. This Cyber Threat Intelligence (CTI) dataset includes 639,450 records from 612 security reports published between January 2008 and June 2019. We covert this dataset to a HIN by following the methodology described [7]. This dataset includes four types of nodes: IP Address (I), Domain Name (D), Malware Hash (M) and Email Address (E), and five types of relationships (edge types): "domain name is resolved to ip address (D–I)", "domain name is visited by malware hash (D–M)", "domain name is registered by email address (D–E)", and "ip address connects to an email address (I–E)". We involve 7 types of meta-paths for the random walks. We assign each month's new nodes and edges to a new time epoch based on combinations of a complete security report.

5. https://dblp.uni-trier.de/

6. The eight categories include: Computational Linguistics, Computer Graphics, Computer Networks and Wireless Communication, Computer Vision and Pattern Recognition, Computing Systems, Databases and Information Systems, Human Computer Interaction, Theoretical Computer Science.

7. https://scholar.google.com/citations?view_op=top_venues&hl=en&vq=eng

6.4 Competitive Methods

To evaluate the advancement of LIME, we firstly compare HETERSNN with the NRL methods below:

Metapath2Vec. This is the state-of-the-art NRL method for HINs [13]. It leverages predefined meta-path-guided random walks to construct the heterogeneous neighborhood of a node and then applies RNN-based skip-gram with negative sampling technology to perform node embedding.

DeepWalk. This method [11] learns d-dimensional vectors by capturing node pairs within w-hop neighborhood via uniform random walks. It is a typical homogeneous network embedding model.

Node2Vec. This model is generalized from DeepWalk, which learns d-dimensional node vectors by capturing node pairs within w-hop neighborhood via parameterized random walks [12]. In this work, we use the suggested parameters, $p = 4$ and $q = 1$, given in the source publication for comparison.

LINE. This model preserves first-order and second-order proximities between nodes [8]. We use two d=2-dimensional vector representations for the first-order and second-order proximities, and then concatenate them as the node representation.

GraphSAGE. This is a inductive NRL framework for dynamic homogeneous networks [27]. It samples node neighborhoods to generate vertex embeddings for unseen data.

PTE. This approach [28] decomposes a HIN to a set of bipartite networks by edge types and learns d-dimensional node vectors by capturing 1-hop neighborhood of the resulting bipartite networks.

We further compare HETERSNN++ against the following 5 representative dynamic NRL methods:

HTNE-a. This employs a multivariate hawkes process and attention mechanism to learn homogeneous temporal network embeddings [18].

DyRep. This method employs time-scale dependent multivariate point process model to learn homogeneous temporal network embeddings [20].

DNE. This method extends homogeneous network embedding methods built around RNN-based skip-gram models to handle dynamic networks [23].

CTDNE. This employs temporal random walk-based continuous-time dynamic network embedding to learn homogeneous and time-preserving network representations [22].

DyHAN. This method uses node-, edge- and temporal-level attention to learn dynamic heterogeneous network embedding [29].

HeterRNN. This is a variant of HETERRSNN. Instead of using the RsNN, it uses an RNN to learn component embeddings.

6.5 Evaluation Methodology

We evaluate all approaches for efficiency and effectiveness.

Efficiency. We measure the resource usage (particularly the memory footprint), execution time and the speedup of training time in both static and dynamic network scenarios. To compute the speedup, we use metapath2vec as the baseline.

Effectiveness. We evaluate the quality of the learned node representation by feeding it to standard machine learning tools to support downstream processing tasks. We consider three tasks: node clustering, node classification, and anomaly detection. For node clustering, we apply k-means to scholar networks to group nodes of authors and venues, and use normalized mutual information (NMI) [39] to quantify the clustering effectiveness. For node classification, we consider both multi-class and multi-label classification. We feed the embedding results into a logistic regression classifier and use Macro-F1 and Micro-F1, two high-is-better metrics [40], to evaluate the classification performance. For anomaly detection, we apply a logistic regression classifier to perform node classification on the CTI dataset and use Macro-F1 and Micro-F1 to evaluate the performance. For the three tasks, we set the embedding size to 129 (which is a multiple of 3 as required by LIME's 3-component representation). To observe the convergence of error and bias in our SGD solution, we also measure the difference of the objective function (error) under different incremental scenarios. This indicates the bias introduced from the parameter inheritance.

Performance report. To minimize the noise, we repeat each experiment 10 times independently and compute the average running time or accuracy. For a fair comparison, we vary the hyper-parameters for each competing method for each task, and use the best-performing settings.

7 EXPERIMENTAL RESULTS

This section demonstrates that LIME is efficient by using the least amount of memory resources and computational time (§ 7.1), and effective by delivering comparable or even better performance for downstream processing tasks (§ 7.2).

7.1 Efficiency Evaluation

In this experiment, we apply NRL methods to the DBLP dataset in both static and dynamic settings. To evaluate the impact of network size on resource and computational efficiencies, we sample the entire DBLP datasets to construct networks of different sizes. Specifically, we choose a subset of publication venues, which are then used to choose the associated nodes like authors and organizations. We use five sample rates to choose the venues, 20%, 40%, 60%, 80%, and 100%, and a sample rate of 100% means we use the entire DBLP dataset. We empirically use the meta-path "O-A-P-V-P-A-O" to guide random walks in the dataset. We set the dimension of node embedding vectors to be 513 (so that it can be divide exactly by 3 since LIME uses a 3-component vector representation).

TABLE I
Time and memory overhead for different sized DBLP networks.

	Method	20%	40%	60%	80%	100%
Time (min.)	DeepWalk	13.802	35.003	39.106	41.428	42.059
	Node2vec	13.815	35.218	39.304	41.694	42.260
	LINE	6.2814	17.005	19.463	21.856	22.352
	PTE	22.736	54.504	58.659	62.142	63.088
	Metapath2v.	13.813	35.491	39.137	41.781	42.402
	GraphSAGE	41.255	85.801	130.588	180.502	226.104
	LIME	6.109	16.190	17.809	19.192	20.056
Mem. (GB)	DeepWalk	2.918	4.041	6.928	8.817	10.437
	Node2vec	2.955	4.058	6.931	8.827	10.445
	LINE	2.954	4.060	6.933	8.833	10.452
	PTE	2.965	4.088	6.952	8.840	10.463
	Metapath2v.	2.998	4.020	6.970	8.833	10.190
	GraphSAGE	3.905	5.911	8.148	10.663	13.021
	LIME	0.712	0.793	0.859	0.904	0.983

7.1.1 Static embedding learning

Table I reports the memory footprint and training time for learning node embeddings for a static DLBP dataset. LIME delivers the fastest training time by using the least amount of memory. Metapath2Vec is the best-performing alternative method for HIN embedding, but it requires at least 4x (up to 6.3x) more memory space and is 2x slower than LIME. It is to note that some memory space (i.e., 0.7GB to 0.9GB) is used to store the node sequences generated by random walks – this is a cost must be paid by all methods. By excluding this overhead, LIME only requires less than 90MB to store the node vectors. By comparison, alternative methods at least 2GB (up to 10GB) for storing the node vectors, which thus incurs significantly larger memory overhead. By using a fewer number of node vectors, LIME also speeds up the training time considerably, because doing so also reduces the number of computational operations and memory accesses. DeepWalk, Node2Vec, LINE, PTE and Metapath2Vec have similar time and space consumption due to their similar inherent mechanism of random walk and negative sampling. Overall, LIME reduces the memory footprint required for learning static network embeddings by at least 4x (up to 10x) and speeds up the training time by at least 2x (up to 16x) over competing methods. This lower memory resource requirement and faster processing time mean that LIME can scale better to larger network for a given processing hardware platform.

7.1.2 Dynamic embedding learning

In this experiment, we consider the DBLP data generated between 1936 to 2017 as the starting, orgenesis network, i.e., $G(t); t = 0$. We then incrementally add data from January 2018 onwards into the genesis network on a daily, weekly, monthly, seasonally, and yearly basis, to form a changing network, $G(t + 1)$, with different time scales. The average number of new nodes introduced by the different time scales, i.e., the daily, weekly, monthly, seasonally, and yearly basis, is 1,753, 11,255, 42,960, 108,983, and 304,633, respectively. The changed network is used to evaluate the

Fig. 5. Comparing LIME's incremental learning strategy against the best alternative static embedding method and the mainstream learning models for dynamic network representation. The x-axis shows the granularity for applying incremental learning to a changed network. LIME is signi cantly faster in learning embeddings for an evolving network.

performance of our incremental learning strategy (x5). As no prior work in heterogeneous information network embedding supports effective incremental learning, we adopt Metapath2Vec– the best performer of prior work in static embedding learning – together with other 5 representative baselines to evaluate the performance of dynamic NRL. Speci cally, we compare to the 'cold-start' Metapath2Vecthat learns embeddings from scratch on a changed network.

Fig. 5 depicts the comparison of raw processing time. By leveraging the changing network structures and the previously learned embeddings of the target network, LIME accelerates the learning time by 5 20x compared with the Metapath2Vec It is worth noting that the processing time of Metapath2Vecis relatively stable because of the number of new nodes is small compared to the ones in the genesis network. We can observe that LIME has a larger advantage over Metapath2Vecwhen processing under a smaller changing scale (e.g., one day or one week). This is because the fewer the changes in the network, the smaller number of optimization operations needed to be performed during the incremental learning process. We consider this a bene t of LIME as it allows us to update node representation more frequently so that the downstream processing model can catch up with the evolution of a network quicker. Compared with continuous random walk based approaches such as CTDNE and DNE, LIME signi cantly bene ts from the shared vector mechanism and thus obtains higher computational ef- ciency. DyRep and HTNE-a models spend more time on modeling the temporal process and DyHAN consumes the longest time because the 3-layer self-attention units are extremely computation intensive, making it unrealistic for handling evolving networks.

7.2 Effectiveness Evaluation

We now evaluate the quality of the learned representation by feeding the node embeddings to standard machine learning algorithms for three downstream processing tasks: node clustering, node classi cation, and anomaly detection.

7.2.1 Node clustering

In this experiment, we use nodes of authors and publication venues gathered the eight research elds (categories) in the AMiner dataset to evaluate how the representation learned by different embedding methods perform on node clustering. The goal of node clustering is to group authors and venues in the same research eld into the same cluster. To label the research eld of an author, we select the category in which the author has the most article records.

TABLE II
Normalized mutual information (NMI) for node clustering. This is a higher-is-better metric.

	Methods	Author Cluster.	Venue Cluster.
Static	DeepWalk	0.4941	0.8521
	Node2vec	0.6246	0.8902
	LINE	0.6423	0.8967
	GraphSAGE	0.6479	0.9003
	PTE	0.6483	0.9060
	Metapath2vec	0.7470	0.9274
	HeterRNN	0.6450	0.8906
	HeterRsNN (LIME)	0.7794	0.9356
Dyn.	DyRep	0.5483	0.8801
	CTDNE	0.6873	0.9104
	DyHAN	0.6628	0.9035
	HTNE-a	0.5248	0.8717
	DNE	0.6538	0.8813
	HeterRsNN++ (LIME)	0.7685	0.9306

In addition to static network embedding models, we also compare our approach against other dynamic NRL baselines. We use 80% of venues related node and edges as the starting network and the remains as new nodes added into the initial network. We apply the k-means algorithm (k = 8 as we target eight research elds) to cluster the author and venue nodes using the learned node embeddings. We then evaluate the performance of the clustering results by computing the NMI.

Table II gives the NMI scores for static and dynamic learning. While using signi cantly fewer number of node vectors, HeterRsNN and HeterRsNN++ employed by LIME outperform all other comparative methods. This is because LIME can better discover the internal semantic relationships among nodes in the HIN due to its novel node placement optimization scheme (x 4.3 and x 5.3). When performing author clustering for the static network, HeterRsNN gives an improvement of 3% on NMI over the best-performing comparative method, Metapath2vec and an improvement of 13%-28% over others. For author clustering in the dynamic network, improves HTNE-a and DNE by 9% and 22% respectively. For venue clustering, our approach also gives the highest NMI score, with a 1%-8% improvement over others. Considering LIME is designed to trade node embedding quality for reduced computational resources, any improvement it achieves would be a bonus. Owing to the bene cial strategy of grouping nodes that may share similar semantics together, LIME achieves the best performance in node clustering tasks. Therefore, the improved NMI score given by LIME is remarkable.

7.2.2 Node classi cation

This task predicts which of the eight research elds in the AMiner dataset, an author or venue node is likely to be based on other training labeled nodes. As an author can contribute into multiple research elds, we formulate the author node classicization a multi-label classi cation

TABLE III
Multi-label author node classi cation in AMiner.

			10%	30%	50%	70%	90%
Macro-F1	Static	DeepWalk	0.6341	0.6684	0.6687	0.6785	0.6888
		Node2vec	0.6365	0.6614	0.6633	0.6738	0.6848
		LINE	0.6486	0.6711	0.6714	0.6726	0.6883
		GraphSAGE	0.6715	0.6724	0.6759	0.6786	0.6815
		PTE	0.6536	0.6817	0.6854	0.6892	0.6944
		Metapath2vec	0.6863	0.7149	0.7185	0.7211	0.7327
		HeterRNN	0.6387	0.6691	0.6677	0.6727	0.6895
		HETERRSNN	0.6908	0.7135	0.7237	0.7249	0.7242
	Dyn.	DyRep	0.6558	0.6829	0.6874	0.6893	0.6988
		CTDNE	0.6520	0.6806	0.6838	0.6839	0.6961
		DyHAN	0.6753	0.6784	0.6773	0.6791	0.6836
		HTNE-a	0.6573	0.6819	0.6834	0.6958	0.6991
		DNE	0.6527	0.6744	0.6715	0.6747	0.6910
		HeterRsNN++	0.6856	0.7134	0.7195	0.7253	0.7322
Micro-F1	Static	DeepWalk	0.6486	0.6787	0.6789	0.6887	0.6973
		Node2vec	0.6493	0.6737	0.6767	0.6829	0.6755
		LINE	0.6573	0.6834	0.6845	0.6878	0.6966
		GraphSAGE	0.6604	0.6833	0.6850	0.6883	0.6995
		PTE	0.6642	0.6916	0.6968	0.6977	0.7082
		Metapath2vec	0.6919	0.7256	0.7265	0.7325	0.7369
		HeterRNN	0.6522	0.6817	0.6810	0.6814	0.6809
		HETERRSNN	0.6964	0.7249	0.7291	0.7374	0.7221
	Dyn.	DyRep	0.6660	0.6927	0.6984	0.6991	0.7093
		CTDNE	0.6651	0.6919	0.6952	0.6972	0.7064
		DyHAN	0.6682	0.6874	0.6904	0.6914	0.7199
		HTNE-a	0.6681	0.6924	0.6909	0.6957	0.7005
		DNE	0.6581	0.6895	0.6900	0.6903	0.7973
		HETERRSNN++	0.6911	0.7238	0.7257	0.7310	0.7218

TABLE IV
Multi-class venue node classi cation in AMiner.

			10%	30%	50%	70%	90%
Macro-F1	Static	DeepWalk	0.3796	0.6695	0.7812	0.8674	0.8357
		Node2vec	0.4486	0.7767	0.8394	0.8935	0.9177
		LINE	0.4629	0.8473	0.9203	0.9466	0.9466
		GraphSAGE	0.4725	0.8591	0.9271	0.9493	0.9507
		PTE	0.3388	0.8304	0.9210	0.9505	0.9489
		Metapath2vec	0.5247	0.8971	0.9532	0.9701	0.9670
		HeterRNN	0.4481	0.7635	0.8489	0.9067	0.9091
		HETERRSNN	0.5082	0.8735	0.9391	0.9605	0.9527
	Dyn.	DyRep	0.4293	0.8388	0.9267	0.9583	0.9499
		CTDNE	0.4267	0.8341	0.9236	0.9540	0.9482
		DyHAN	0.4884	0.8590	0.9291	0.9593	0.9469
		HTNE-a	0.3984	0.6813	0.7966	0.8795	0.8577
		DNE	0.4612	0.8438	0.9195	0.9482	0.9484
		HETERRSNN++	0.5039	0.8747	0.9385	0.9609	0.9522
Micro-F1	Static	DeepWalk	0.4042	0.7166	0.7990	0.8877	0.9186
		Node2vec	0.4981	0.7957	0.8586	0.9145	0.9451
		LINE	0.5167	0.8457	0.9209	0.9500	0.9571
		GraphSAGE	0.5258	0.8522	0.9281	0.9569	0.9611
		PTE	0.4267	0.8372	0.9239	0.9550	0.9571
		Metapath2vec	0.5975	0.9011	0.9522	0.9725	0.9857
		HeterRNN	0.4974	0.7939	0.8591	0.9187	0.9301
		HETERRSNN	0.5751	0.8829	0.9389	0.9605	0.9693
	Dyn.	DyRep	0.4305	0.8413	0.9280	0.9595	0.9509
		CTDNE	0.4392	0.8504	0.9253	0.9521	0.9568
		DyHAN	0.5133	0.8654	0.9308	0.9510	0.9687
		HTNE-a	0.4274	0.7381	0.8229	0.9011	0.9248
		DNE	0.5138	0.8503	0.9211	0.9489	0.9571
		HETERRSNN++	0.5733	0.8845	0.9377	0.9606	0.9671

problem. Here, we learn node embeddings for all authors and venues and feed the labeled nodes (as introduced in x 6) into a logistic regression classi er. We use 10%, 30%, 50%, 70% to 90% of the data to train the classi er and the remaining for testing. To evaluate dynamic embedding learning, we use the AMiner dataset by a monthly step and compare HeterRsNN++ with dynamic NRL baselines.

For all training-test-split ratios, LIME outperforms most baselines with an average 0.5%-6% improvement for author node classi cation in Table III and an average 1%-17% for venue classi cation in Table IV. In some test settings, LIME gives marginally lower performance (between 0.5% and 2%) over the best-performing HIN NRL method Metapath2vec for static embedding, but with signi cantly less computational resources. The comparable performance suggests that LIME is effective in capturing an ensemble of semantic and structural correlations of heterogeneous networks. The vector sharing mechanism adopted by LIME allows multiple nodes to share component vector(s), which inherently result in an increased correlation among these nodes. If we consider the RNN variant of LIME (e.g., replacing the RsNN with an RNN), we see that HeterRNN does not deliver the state-of-the-art performance, suggesting that an RNN is less effective in modeling network structures. Finally, we also observe that the scores of HeterRsNN (that operates on the entire dataset) and HeterRsNN++ are close. This suggests the effectiveness of incremental learning approach.

TABLE V
Anomaly detection on the CTI dataset.

			10%	30%	50%	70%	90%
Macro-F1	Static	DeepWalk	0.5071	0.6084	0.6219	0.6445	0.6492
		Node2vec	0.5753	0.6621	0.6957	0.7118	0.7245
		LINE	0.5844	0.6669	0.6983	0.7125	0.7353
		GraphSAGE	0.5858	0.6721	0.7005	0.7238	0.7386
		PTE	0.5675	0.6569	0.6848	0.7033	0.7219
		Metapath2vec	0.5882	0.6891	0.7289	0.7400	0.7491
		HeterRNN	0.5685	0.6609	0.6921	0.7107	0.7241
		HETERRSNN	0.5947	0.6915	0.7291	0.7485	0.7522
	Dyn.	DyRep	0.5724	0.6618	0.6882	0.7011	0.7225
		CTDNE	0.5766	0.6659	0.6832	0.7104	0.7306
		DyHAN	0.5856	0.6871	0.7188	0.7332	0.7419
		HTNE-a	0.5622	0.6501	0.6885	0.7011	0.7218
		DNE	0.5823	0.6695	0.6932	0.7119	0.7355
		HETERRSNN++	0.5873	0.6989	0.7286	0.7451	0.7490
Micro-F1	Static	DeepWalk	0.5492	0.6515	0.6841	0.6990	0.7127
		Node2vec	0.6255	0.7227	0.7501	0.7658	0.7713
		LINE	0.6311	0.7425	0.7699	0.7823	0.7894
		GraphSAGE	0.6327	0.7429	0.7701	0.7874	0.7925
		PTE	0.6149	0.7205	0.7433	0.7599	0.7704
		Metapath2vec	0.6455	0.7486	0.7831	0.7979	0.8080
		HeterRNN	0.6128	0.7044	0.7415	0.7532	0.7629
		HETERRSNN	0.6483	0.7502	0.7833	0.7980	0.8035
	Dyn.	DyRep	0.6166	0.7214	0.7433	0.7610	0.7844
		CTDNE	0.6194	0.7201	0.7486	0.7658	0.7894
		DyHAN	0.6314	0.7321	0.7568	0.7658	0.7905
		HTNE-a	0.6175	0.7291	0.7481	0.7618	0.7713
		DNE	0.6321	0.7208	0.7415	0.7558	0.7883
		HETERRSNN++	0.6385	0.7403	0.7786	0.7850	0.7955

7.2.3 Anomaly detection

In this task, we train a logistic regression classi er to take as input the node embeddings to predict potential anomalies of the CTI dataset. Like node classi cation (x7.2.2), we vary the training ratios from 10% to 90% with a step of 20%. Because a security report can be attributed with multiple catalogue tags, we formulate this as a multi-label classi cation problem. Similarly, we grow the CTI dataset by a monthly basis to evaluate dynamic embedding learning.

Table V gives the Macro-F1 and Micro-F1 scores for each approach. Like node classi cation, HETERRSNN and HETERRSNN++ outpeform all but Metapath2vec. However, the performance gap between HETERRSNN/HETERRSNN++ and Metapath2vec is small, between 0.1% and 0.8%. We also observe that HETERRSNN ++ delivers similar performance over HETERRSNN, albeit it applies a incremental learning strategy that has lower memory footprint and faster training

TABLE VI
LIME can automatically group venue/author nodes in the same research eld to share the same row or column vector.

Vector no.	Network nodes (venues or authors)	Research eld
row-45	TC, TPDS, TOS, ASPLOS, IPDPS, TOCS, HPCA, OSDI, SOSP, FAST, ICDCS, TECS, PARCO, ISC	Computing systems
row-72	VLDB, ICDE, SIGMOD, ICDT, TODS, VLDBJ, CIDR, EDBT, PVLDB, PODS, DASFAA, SSDBM	Database
row-86	KDD, TKDE, ICDM, TKDD, WSDM, DMKD, CIKM, SDM, KAIS, ECML-PKDD, PAKDD	Data mining
row-99	Albert Zomaya, Andrew S. Tanenbaum, Ion Stoica, Schahram Dustdar, David A. Patterson	Computing systems
row-121	NIPS, ICML, JMLR, TPAMI, Neural Computation, AIJ, UAI, IJCV, TNNLS, Machine Learning	Machine learning
column-19	Jiawei Han, Philip S. Yu, Christos Faloutsos, Jian Pei, Xindong Wu, Charu Aggarwal	Data mining
column-55	CVPR, ICCV, TPAMI, MM, TIP, ECCV, TMM, IJCV, ACCV, CVIU, IET-CVI	Computer vision
column-89	ACL, EMNLP, NAACL, COLING, TSLP, TASLP, CoNLL, JSLHR, Computational Linguistics	Computational linguistics
column-131	AAAI, IJCAI, UAI, Arti cial Intelligence Journal, JAIR, ECAI, JSLHR, IJAR, Neural Networks	AI
column-71	Andrew Zisserman, Jitendra Malik, Andrew Fitzgibbon, Eric Grimson, Roberto Cipolla	Computer vision

time. This experiment shows that LIME can extract high-quality network representation for anomaly detection.

7.3 Observable Error of Objective Loss Function

To validate the effectiveness of model convergences, we conduct an empirical micro-benchmarking study on the error of objective loss function in HETERRSNN++ and cold-start HETERRSNN, respectively. In general, a smaller loss indicates better model effectiveness of unsupervised learning models. We leverage different datasets with various sizes extracted from DBLP to showcase different time scales and their effects on the dynamic HINs. We con gure the increment on a daily, weekly, monthly, seasonally, and yearly basis, respectively. In Fig. 6, there is an observably stable but descending trend of the objective function differences under different increment scales. The error tends to decrease as the number of HIN nodes soars. For instance, the error of daily increment in HETERRSNN++ is roughly 0.0804 while it falls to merely 0.0004 in case of yearly increment. This is because more HIN nodes are involved in the bias calibration, wherein a increasing number of training iterations will gradually improve the precision despite the inevitably longer processing time (Fig. 5). The result indicates the objective loss error can be constantly maintained at an extremely low level with minimized bias stemming from dynamic network changes.

7.4 Case study

As a signi cant departure from NRL methods [3], LIME can automatically discover the internal semantic relationship among nodes in our cuboid space. In an attempt to illustrate this advantage, we show the author and venue nodes that share the same row and column vectors when applying HETERRSNN to the DBLP dataset. The results are given in Table VI, which include both publication venues and authors of different research elds.

As can be seen from Table VI, LIME is highly effective in grouping nodes with similar semantics. For example, the publication venues in row-45 are all conferences and journals in computing systems. We also observe similar grouping for authors and venues in other elds of data mining, databases, AI, ML, NLP, CV, etc. Given that the grouping is done automatically without human involvement or similar search, LIME is thus able to learn the semantic relationships during training. The results show that our 3-component shared embedding scheme is about to capture and incorporate the underlying structural and semantic relationships between various types of nodes in heterogeneous networks. This allows LIME to reduce the number of node vector without signi cantly compromising the quality of the node representation.

Fig. 6. Objective loss on different sized HIN.

8 Related Work

Deep network embedding models. Previous network representation learning models primarily focus on improving the learning ability such as preserving original network structural information and properties [3] or semantic correlations of different types of nodes and relationships [13], [41], [5]. Unlike these, LIME aims to improve the ef ciency in computational resources and training time to allow the learning algorithm to scale to large networks and to respond to a dynamically changing HIN quickly. While graph-based learning methods have recently demonstrated impressive results on learning graph representations, graph neural networks (GNNs) would incur signi cant memory overhead and long training time for real-life networks. As a result, GNNs are ill-suited for learning embedding for larger, dynamically changing HINs.

LightRNN [25] is the most closely related work for resource-tuned representation learning. It targets word embedding learning for natural language processing tasks. LightRNN adopts a 2-component shared embedding scheme using RNN for learning word embeddings. LightRNN allocates every word in the word vocabulary into a table so that words in a row shard with a row vector and words in each column share a common column vector. While LightRNN reduces the model size and running time for processing texts, a 2-component scheme is

ill-suited for HINs because it cannot adequately model the semantics among heterogeneous nodes in a HIN. Inspired by LightRNN, we adopt a 3-component shared scheme and employ RsNN to propagate and aggregate information across the network hierarchy. LIME also advances LightRNN by using a faster and more accurate algorithm to solve the optimization problem in shared embedding space.

Dynamic network embedding models. Numerous dynamic network embedding models have been proposed and can be classified into three categories: i) Stochastic-based approaches [18], [19], [20] assume the network changes follow a certain stochastic process (e.g., Hawkes process, Triadic closure process, or Multivariate point process), which can hardly stand in realistic networks. ii) Approaches based on temporal random walks [22], [23], [24] exploit snapshot and skip-gram technologies for learning the embedding. However, such models have intrinsic limited scalability in terms of computation and storage, and ignore the heterogeneity of network structure, thereby limiting the representation precision. iii) Attention-based approaches [29] use node-, edge- and temporal-level attention for graph embedding, most suitable for link prediction. However, the computation and space complexity impede its applications in large-scale networks, wherein out-of-memory tends to manifest. Furthermore, other task-specific graph embedding approaches [42], [43] use supervised/semi-supervised modeling to learn dynamic rules. However, they aim to predict the structure of graph instead of effective node embedding, causing prohibitively long training time. One can easily integrate the proposed node embedding LIME with recurrent neural network such as LSTM to effectively fulfill such prediction. As a departure from prior work, LIME has significantly accelerated the embedding procedure with lower memory cost but can achieve higher-quality embeddings.

9 Conclusion

Networks are a universal language for modeling complex systems. The ability for understanding and characterizing network structures underpins many applications. However, realizing this ability in a resource- and time-efficient manner is highly challenging because real-world networks encompass massive heterogeneous nodes and edges and can change drastically over time. We have presented LIME, a fast, resource-efficient method for extracting useful representation from dynamic information networks. To reduce the memory requirement for learning network representation, LIME exploits the semantic relationships among network nodes to encode multiple nodes with similar semantics in shared vectors. By using many fewer node vectors, LIME thus significantly cuts down the memory space and computational time over the state-of-the-arts. To minimize the information lost when using fewer node vectors, LIME exploits the recursive neural network with carefully designed optimization strategies to explore the node semantics in a novel cuboid space. To quickly adapt to the changes in a network, we develop a novel incremental learning for changing networks. We apply LIME to three large-scale datasets across three downstream processing tasks and compare LIME against eight prior state-of-the-art methods for learning network representation. Our extensive experiments show that LIME reduces the memory footprint by over 80% and computational time over 2x, without compromising the quality of the learned network representation.

Acknowledgments

The authors of this paper were supported by the NSFC through grants 62002007 and U20B2053, the Key Research and Development Project of Hebei Province through grant 20310101D, the UK EPSRC (EP/T01461X/1, EP/T021985/1, EP/R033293/1, EP/T022582/1), a UK Royal Society International Collaboration Grant, NERC (NE/P017134/1), Australian Research Council Discovery scheme under grant DP200103494, SKLSDE-2020ZX-12, NSF ONR N00014-18-1-2009, NSF under grants III-1763325, III-1909323, and SaTC-1930941. This work was also sponsored by CAAI-Huawei MindSpore Open Fund. Thanks for computing infrastructure provided by Huawei MindSpore platform.

References

[1] Z. Liu et al., "Alleviating the inconsistency problem of applying graph neural network to fraud detection," in Proceedings of the SIGIR. ACM, 2020, pp. 1569–1572.
[2] Z. Wang et al., "Predictive network representation learning for link prediction," in SIGIR. ACM, 2017, pp. 969–972.
[3] Z. Wu et al., "A comprehensive survey on graph neural networks," IEEE Transactions on Neural Networks and Learning Systems, 2020.
[4] H. Peng et al., "Fine-grained event categorization with heterogeneous graph convolutional networks," in IJCAI, 2019.
[5] Y. He et al., "Hetespaceywalk: A heterogeneous spacey random walk for heterogeneous information network embedding," in Proceedings of the CIKM. ACM, 2019, pp. 639–648.
[6] Y. Cao et al., "Multi-information source hin for medical concept embedding," in Proceedings of the PAKDD. Springer, 2020, pp. 396–408.
[7] Y. Gao et al., "Hincti: A cyber threat intelligence modeling and identification system based on heterogeneous information network," TKDE, 2020.
[8] J. Tang et al., "Line: Large-scale information network embedding," in Proceedings of the WWW. ACM, 2015, pp. 1067–1077.
[9] D. Yogatama et al., "Embedding methods for fine grained entity type classification," in Proceedings of the ACL, 2015, pp. 291–296.
[10] H. Peng et al., "Hierarchical taxonomy-aware and attentional graph capsule rcnns for large-scale multi-label text classification," TKDE, 2019.
[11] B. Perozzi et al., "Deepwalk: online learning of social representations," in Proceedings of the KDD. ACM, 2014, pp. 701–710.
[12] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in Proceedings of the KDD, 2016, pp. 855–864.
[13] Y. Dong et al., "metapath2vec: Scalable representation learning for heterogeneous networks," in Proceedings of the KDD, 2017.
[14] D. Wang et al., "Structural deep network embedding," in Proceedings of the KDD, 2016, pp. 1225–1234.
[15] S. Cao et al., "Grarep: Learning graph representations with global structural information," in Proceedings of the CIKM, 2015, pp. 891–900.
[16] F. Scarselli et al., "The graph neural network model," IEEE Transactions on Neural Networks, vol. 20, no. 1, pp. 61–80, 2008.
[17] K. Xu et al., "How powerful are graph neural networks?" in ICLR, 2019.
[18] Y. Zuo et al., "Embedding temporal network via neighborhood formation," in Proceedings of the KDD, 2018, pp. 2857–2866.
[19] L. Zhou et al., "Dynamic Network Embedding by Modelling Triadic Closure Process," in AAAI, 2018.
[20] R. Trivedi et al., "Representation learning over dynamic graphs," in ICLR, 2019.
[21] H. Peng et al., "Dynamic network embedding via incremental skip-gram with negative sampling," Science China Information Sciences, vol. 63, no. 10, pp. 1–19, 2020.
[22] G. H. Nguyen et al., "Continuous-time dynamic network embeddings," in Proceedings of the WWW, 2018, pp. 969–976.