This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

DyGCN: Efficient Dynamic Graph Embedding With Graph Convolutional Network

Zeyu Cui, Zekun Li, Shu Wu, *Senior Member, IEEE*, Xiaoyu Zhang, *Senior Member, IEEE*, Qiang Liu, *Member, IEEE*, Liang Wang, *Fellow, IEEE*, and Mengmeng Ai

***Abstract*— Graph embedding, aiming to learn low-dimensional representations (aka. embeddings) of nodes in graphs, has received significant attention. In recent years, there has been a surge of efforts, among which graph convolutional networks (GCNs) have emerged as an effective class of models. However, these methods mainly focus on the static graph embedding. In the present work, an efficient dynamic graph embedding approach is proposed, called dynamic GCN (DyGCN), which is an extension of the GCN-based methods. The embedding propagation scheme of GCN is naturally generalized to a dynamic setting in an efficient manner, which propagates the change in topological structure and neighborhood embeddings along the graph to update the node embeddings. The most affected nodes are updated first, and then their changes are propagated to further nodes, which in turn are updated. Extensive experiments on various dynamic graphs showed that the proposed model can update the node embeddings in a time-saving and performance-preserving way.**

***Index Terms*— Dynamic graphs, graph convolutional network (GCN), neural network.**

I. Introduction

GRAPHS are natural representations for encoding relational structures and therefore have a wide range of applications in bioinformatics [14], chemistry [1], recommender systems [16], and social network studies [29], among others. However, unlike images that are in a compact grid pattern, the graph structure presents a topological pattern, which is not easy to process by common machine learning methods. Although graphs can be presented in the form of an adjacency matrix, they are too big to save or process when the number of nodes is increased.

Graph embedding, finding a mapping function to transform each node in the graph into a low-dimensional latent representation, is considered as an effective way to represent a graph and has attracted considerable research efforts [12], [31], [37]. However, most researches focus on static graph embedding, i.e., learning representations only of static graphs. Three popular classes of graph embedding have emerged in recent years, some of which are based on matrix factorization (MF) [3], [4], [37]. Belkin et al. [3] used singular value decomposition (SVD) to decompose the node representation, whereas large-scale information network embedding (LINE) [37] applied factorization strategies into large-scale graph embedding. Some graph embedding methods were inspired by the random walk strategy, including DeepWalk [31] and Node2Vec [12]. These random-walk-based methods transform the graph structure into several random walk sequences, which can be easily processed by the skip-gram model in Word2Vec [28]. Graph convolution networks (GCNs) [9], [13], [18], [35], [48], as a class of models generalizing neural networks to graph-structured data, have recently gained significant attention. The GCN-based methods generally follow an embedding propagation scheme, in which the embedding of each node is recursively updated by the aggregating message propagated from its neighboring nodes [48]. Under the embedding propagation scheme, the GCN-based methods have achieved significant improvement in terms of both efficiency and effectiveness.

Many graphs in the real world are inherently dynamic, meaning that their nodes/edges may constantly change over time. For instance, on an online platform, users may join or leave a social network at any time and may develop new relationships or break off other relationships over time. Users may also change their profile information, such as age and location. Thus, considering the graph embedding problem in a dynamic manner is more appropriate for real-world applications. More information can be captured when the dynamic features of the graph are considered [17]. Although some researches on dynamic graph embedding have been proposed [11], [34], [38], [54], they mainly focused on mining the pattern of graph evolvement but ignored the efficiency issue. Their methods consequently achieved a better performance than previous static methods. However, generating a new representation at each time is costly, given that most

Manuscript received 23 August 2021; revised 29 March 2022; accepted 12 June 2022. This work was supported in part by the National Key Research and Development Program under Grant 2019QY1601 and in part by the National Natural Science Foundation of China under Grant 62141608 and Grant U19B2038. *(Zeyu Cui and Zekun Li contributed equally to this work.) (Corresponding author: Shu.Wu.)*

Zeyu Cui is with the Alibaba DAMO Academy, Jinhui Building, Beijing 100102, China (e-mail: cuizeyu15@gmail.com).

Zekun Li is with the Department of Computer Science, University of California at Santa Barbara, Santa Barbara, CA 93106 USA.

Shu Wu is with the Institute of Automation, Chinese Academy of Sciences, Beijing 100098, China.

Xiaoyu Zhang is with the Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China.

Qiang Liu is with the Institute of Automation, Chinese Academy of Sciences, Beijing 100098, China.

Liang Wang is with NLPR, CASIA, Beijing, China.

Mengmeng Ai is with the International School, Beijing University of Posts and Telecommunications, Beijing 100876, China.

Data is available on-line at https://github.com/CRIPAC-DIG/DyGCN.

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TNNLS.2022.3185527.

Digital Object Identifier 10.1109/TNNLS.2022.3185527

2162-237X © 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: INSTITUTE OF AUTOMATION CAS. Downloaded on November 03,2022 at 06:50:37 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

traditional node representation methods are designed to learn the embedding parameters through an optimization process (e.g., gradient descent or matrix factorization). Repeating these processes at each time step results in high complexity.

The time complexity problem originates mainly from two aspects. First, global update time-consuming, that is to update all the node embeddings at each time step. Second, retraining time-consuming, that is to update node embeddings via optimization progress, e.g., matrix factorization or stochastic gradient descent. However, both the aspects of time consumption can be reduced by leveraging the dynamic relationships between consequent time steps. Because changes in graphs are frequent and trivial at any time, the node embedding may not change considerably; hence, it may be unnecessary to update the embeddings of all the nodes at each time step. Some researchers thus try to avoid the recalculation of all node representations and consider only the changing nodes, e.g., DNE [7], DynWalks [15], and NetWalk [50]. Some, such as TIMERS [52] and Dyrep [41], avoid the relearning process and directly update the previous embeddings according to the changes.

GCN has recently shown great power in graph embedding and has the potential to deal with graphs incrementally [13]. Therefore, this research aims to extend GCN to the dynamic environment in an efficient manner.

Present Work: In this study, an efficient dynamic graph embedding framework called dynamic graph convolutional network (DyGCN) is proposed, which extends the GCN-based embedding models to dynamic settings. The proposed method may be applicable to all other methods based on GCN, As the core of DyGCN, a dynamic graph convolution operation is devised, which naturally generalizes the embedding propagation scheme of GCN to the dynamic setting, propagating the change along the graph to update the embeddings of the affected nodes. The process consists of two steps: the change in the message aggregated from neighbors is first calculated, and then the node embeddings are adjusted accordingly. The most affected nodes are updated first. Then, the change is propagated to the neighboring nodes, resulting in their update. Two different ways of node propagation are designed: high-order update and spectral propagation. The former is more efficient, but the latter is more precise. Both the ways decrease the computational time compared with other methods of dynamic graph embedding. Extensive experiments on three real-world datasets are carried out. The experimental results show the efficiency and effectiveness of the proposed method compared with other dynamic graph embedding methods and the static counterpart (GCN).

Overall, the contributions of this work can be summarized as follows.

1) The proposal of DyGCN as an efficient dynamic embedding framework for the GCN-based embedding methods, which is capable of updating node embeddings in dynamic graphs in a time-saving and performance-preserving way.
2) The creation of a dynamic graph convolution operation as the core of DyGCN, which efficiently propagates the change along the graph to update the node embeddings.
3) Extensive experiments on various datasets to verify the effectiveness and efficiency of the proposed approach.

II. Related Works

In this section, three streams of research related to the present work are briefly reviewed, i.e., graph embedding methods, dynamic graph embedding methods, and graph neural networks (GNNs). Note that we only discuss the unsupervised graph embedding methods in this brief.

A. Graph Embedding Methods

Graph embedding (also called network embedding) is an important research area in graph analysis. Originally, a typical task of the dimension reduction problem, graph embedding is defined as representing an $n \times n$ adjacency matrix as an $n \times k$ matrix, where $k \ll n$. Some representative methods in this line include principal component analysis (PCA) [44] and multidimensional scaling (MDS) [20]. Afterward, some methods such as ISOmap [39] and local linear embeddings (LLE) [33] were later designed for better global structure preservation. For larger scale graphs, factorization methods, such as GraRep [4] and LINE [37], are commonly used to apply graph embedding to larger datasets.

In recent years, deep learning methods have already achieved promising results in the field of network embedding areas. The first deep learning method of network embedding is commonly recognized as DeepWalk [31]. After the success of DeepWalk, more deep learning methods for network embeddings have been designed. Node2Vec [12] improved the random walk strategies of DeepWalk by a controllable deep or wide walking possibility. SDNE [43] introduced a deep autoencoder model to learn the 1st- and 2nd-order neighbors of nodes. Thus far, GNNs have received considerable attention, and several frameworks have been developed, including VGAE [19], GraphSAGE, and GAT [42].

B. Dynamic Graph Embedding Methods

There are two streams of research on dynamic graphs. The first, called dynamic graph prediction, aims to predict the graph structure at the next time step [30], [36], [49]; the second, called dynamic graph embedding, tries to *learn* better graph representation according to dynamic information. The present work belongs to the latter stream. The two tasks, graph prediction and graph embedding, can be formulated as follows:

$$\mathcal{G}^{T+1} = f\left(\mathcal{G}^{\leq T}\right) \tag{1}$$

$$X^T = f\left(\mathcal{G}^{\leq T}\right) \tag{2}$$

where $\mathcal{G}^T$ is the graph structure at time step T. X^T is the embedding at time step T. Because this work aims to learn representations for dynamic graphs in an efficient manner, some graph prediction methods, such as EvolveGCN [30] and DynGraph2Vec [10], are not discussed here.

Dynamic graph embedding was initially considered as an extension of static graph embedding. Researchers simply used the static network embedding methods in each snapshot during each time step, with the constraint of aligning the same nodes

Authorized licensed use limited to: INSTITUTE OF AUTOMATION CAS. Downloaded on November 03,2022 at 06:50:37 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

in different time steps. The dynamic graph embedding methods are argued to obtain better representations than the traditional static graph embedding because different snapshots share more characteristics for representation [10], [11], [34], [54], [55].

An inevitable problem with the above methods is that although they are able to learn better representation than the static methods, their computational complexity is unacceptable in real-world applications. For efficiency, researchers try to update only a part of the embeddings of previous snapshots instead of recalculating the current graph totally. TIMERS [52] proposes an incremental SVD model for dynamic embedding, which requires SVD only at the beginning and then incrementally updates them according to the changes on the graph. DNE [7] proposes a dynamic version of LINE, with only a few gradient descent process to update the representation of the current graph. DANE [22] proposes a network embedding method in a dynamic environment that updates the node embedding based on the change in both the adjacency and the attribute matrix through matrix perturbation. DyRep [41] ingests dynamic graph information in the form of association and communication events over time and updates the node representations as they appear in these events. DynWalks [15] incorporates the temporal information with the traditional DeepWalk to capture the evolving properties in dynamic networks. It updates the node embeddings by sampling new walks that are highly related to the changes on the graph.

C. Graph Neural Networks

GNNs, which extend deep neural networks to graph-structured data, are a class of models recently advanced in graph representation learning. Due to their convincing performance and high interpretability, GNNs have been widely used in many applications, such as natural language processing [2], image classification [26], recommendation [46], and fashion analysis [6]. The concept of GNN was first proposed by Scarselli et al. [35]. Most GNN models follow a similar scheme. In general, the nodes in GNNs aggregate information from neighborhoods and update their embeddings iteratively. Various kinds of GNNs have recently been proposed. GCNs [18] perform graph convolutions for aggregation and update motivated by spectral convolution. Gated GNNs (GGNNs) [23] aggregate and update in the form of gated recurrent units (GRUs). Graph attention networks (GATs) [42] incorporate the attention mechanism into the aggregation step. GraphSAGE [13] considers from the spatial perspective and introduces an inductive learning method. After that, researchers focus on the effectiveness and efficiency of the aggregation and updating part [5], [45], [48]. Most of the existing GNN models are designed for the static graph setting, in which nodes and edges are fixed. However, the idea of representing nodes by the aggregation information of neighboring nodes may also be suitable to the dynamic setting.

III. Preliminaries

In this section, the traditional GCN is described in detail. Then, the graph embedding problem is formulated from static and dynamic styles.

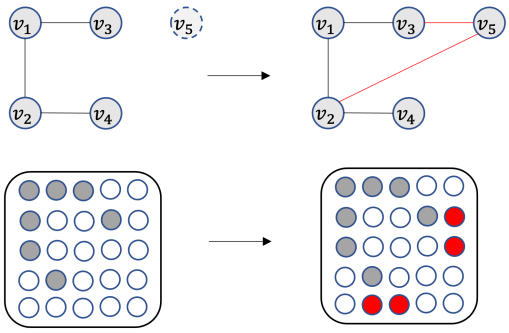

Fig. 1. Illustration of how we deal with the addition/removal of nodes and edges in a unified way (assume there are at most five nodes through all the time steps). The top image shows the change in the graph structure; the bottom image shows the corresponding change in the adjacency matrix. The red lines denote the newly added edges; the red circles in the adjacency matrix denote the newly emerging nonzero entries.

A. Static Graph Embedding

Consider a graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$, $\mathcal{V} = [v_1, v_2, \ldots, v_N]$, and $\mathcal{E} = [e_1, e_2, \ldots, e_M]$ denote the node and edge set in $\mathcal{G}$, respectively. N is the number of nodes and M is the number of edges. We denote the attributes of nodes as X, where x_v represents the attribute of node v in $\mathcal{V}$. The attributes include the node degree and other information based on different kinds of datasets. The adjacency matrix is denoted as A. The static graph embedding methods try to transform A (or A, X) to node embedding matrix $Z \in \mathbb{R}^{N\times d}$

$$Z = \text{Static\_GE}(A, X) \tag{3}$$

where d is the dimension of the embeddings and is much smaller than the number of nodes N. Each row of Z, z_v denotes the representation of node v. The goal of the static graph embedding methods is to obtain a better z_v that can express both the structural and attribute features of node v for downstream tasks, such as node classification and link prediction.

B. Dynamic Graph Embedding

In contrast to the static graph embedding problem, dynamic graph embedding introduces the time step in the formulation. Henceforth, we use the superscript t to denote the time index. In that case, we have a graph sequence $[\mathcal{G}^1, \mathcal{G}^2, \ldots, \mathcal{G}^t, \ldots, \mathcal{G}^T]$, where $\mathcal{G}^t = \{\mathcal{V}^t, \mathcal{E}^t\}$ represents the graph at time step t. T is the length of the sequences. $\mathcal{V}^t$ and $\mathcal{E}^t$ are the set of nodes and edges in $\mathcal{G}^t$, respectively. Without loss of generality, we assume that the graph at any time is assumed to be built on a common node set of cardinality N, with N not lesser than the maximum number of nodes appearing in a single time step. As shown in Fig. 1, the nonexistent node v_5 is treated as a dangling node with zero degree. If a node is added/removed, its corresponding edges to other nodes are added/removed. In this case, the changes in nodes can be seen as the changes in edges, and there would be no need to change the adjacency matrix at each time step. Therefore, the rest of this work mainly talks about the changes in edges.

Given that the node feature matrix X is commonly stable, the change in the topological structure is mainly discussed, i.e., adjacency matrix $A^t = \{0, 1\}^{N\times N}$ in this work. $Z^t \in \mathbb{R}^{N\times d}$

Authorized licensed use limited to: INSTITUTE OF AUTOMATION CAS. Downloaded on November 03,2022 at 06:50:37 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

is the node embeddings' matrix, and each row of which is a d-dimensional embedding of a node in $\mathcal{V}^t$. The dynamic graph embedding methods can be formulated as

$$Z^t = \text{Dyn\_GE}\left(A^{\leq t}, X\right) \tag{4}$$

where $A^{\leq t}$ denotes the adjacency matrices from the beginning to time step t.

There are two different research topics in the area of dynamic graph embedding. The first focuses on obtaining more representative embeddings by considering previous graph snapshots. Theoretically, the lower bound of the representation capacity is that of the static graph embedding representation. The second focuses on the efficiency of learning dynamic graph representation, in which the static graph embedding representation is the upper bound of the representation capacity. In the present research, the main idea for reducing the computation is to update the previous node embeddings based on ΔA^t instead of relearning all the node embeddings at each time step, which is formulated as

$$Z^t = \text{Dyn\_GCN}\left(\Delta A^{t-1}, Z^{t-1}\right) \tag{5}$$

where the initial matrix embedding Z^0 is obtained by the static graph embedding method, and any dynamic graph methods can be applied. $\Delta A^{t-1} = A^t - A^{t-1}$ is the change in the graph structure.

C. Graph Convolutional Network

GCN [13], [18], [42] are a class of models that extend the neural networks to graphs. They take the adjacency matrix A and the node features matrix X as inputs and output the node embedding matrix $Z \in \mathbb{R}^{n\times d}$, where n is the number of nodes and d is the dimensionality of node embeddings

$$Z = \text{GCN}(A, X). \tag{6}$$

A GCN usually consists of multiple layers of graph convolution to iteratively update the node embeddings. Specifically, the node embeddings Z can be updated to Z' through a layer of graph convolution as (the layer index is omitted for clarity)

$$\begin{aligned} Z' &= \text{GCONV}(A, Z, W) \\ &:= \sigma\left(\hat{A}ZW\right) \end{aligned} \tag{7}$$

where $\hat{A}$ denotes some normalization of the graph adjacency matrix. For simplicity, $\hat{A} = A + I$ is used in this work. I is the identity matrix, W is a layer-specific transformation matrix, and σ is a nonlinear activation such as ReLU.

The graph convolution is actually an embedding propagation operation motivated by spectral convolution. Here, 7 is reformulated from the perspective of a single node v, which is totally the same operation as the former equation. For notional clarity, we denote the embedding of a node v is denoted as z_v, which is a row vector of Z. A node z first aggregates the message propagated from neighbors and itself

$$a_v = \sum_{u\in\mathcal{N}(v)\cup v} z_u \tag{8}$$

where a_v is the aggregated information of node v, and $\mathcal{N}(v)$ is the set of its neighbors. Then it generates a new embedding z'_v based on the aggregated message as

$$z'_v = \sigma(a_v W). \tag{9}$$

The initial node embeddings are input node features X. After limited periods of graph convolution, the node embeddings will capture the information propagated from various orders away and output the final node embedding Z.

IV. Proposed Method

In this section, a naive graph convolution network for graph embedding is first presented. Then, two versions of the proposed DyGCN model extending static GCN to the dynamic setting are introduced. Finally, the proposed methods are compared with other methods in terms of effectiveness and efficiency.

A. Naive GCN for Dynamic Graph Embedding

A naive way to use GCN in this dynamic setting is through its direct application at each snapshot to learn the new node embeddings

$$Z^{t+1} = \text{GCN}\left(A^{t+1}, X\right). \tag{10}$$

However, this is costly and unnecessary given that the change at each time step is usually trivial. In particular, there is little difference between A^{t+1} and A^t, or in other words, that is, ΔA^t is sparse.

B. Dynamic GCN

In this section, the proposed DyGCN model is elaborated. Fig. 2 shows the model framework; on the left is an example of a dynamic graph, with an edge between nodes v_1 and v_2 emerging at time t. The change is propagated along the graph to update the node embeddings in turn. v_1 and v_2 are denoted as the 1st-order influenced nodes, being directly influenced by the emerging edge. v_3 and v_4, as well as v_5 and v_6, as the neighbors of the 1st-order influenced nodes v_1 and v_2, are denoted as the 2nd-order influenced nodes. Accordingly, kth-order influenced nodes refer to the nodes that are $(k-1)$ hops away from the 1st-order influenced nodes. The set of the kth-order influenced nodes at time step t is denoted as V_k^t.

The proposed model is based on the embedding propagation scheme of GCN [13], [18], [48]. The node embeddings are updated according to the change in the aggregated message. In the 1st-order propagation layer, the 1st-order influenced nodes are updated; these are the nodes directly connected to the changing edges. The updated embeddings are then propagated to their neighbors, which in turn are updated in the successive high-order propagation layers. Their design is elaborated in the following.

1) First-Order Update: Here, the method of updating the embeddings of the 1st-order influenced nodes is discussed. First, let us review the aggregation and update scheme of GCN in (8) and (9). A node first aggregates the message propagated from its neighbors and itself by summation and then updates the embeddings based on it. The aggregated message determines the final embeddings. The change in the topological structure influences the aggregated message and, thus, the final node embeddings.

Authorized licensed use limited to: INSTITUTE OF AUTOMATION CAS. Downloaded on November 03,2022 at 06:50:37 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

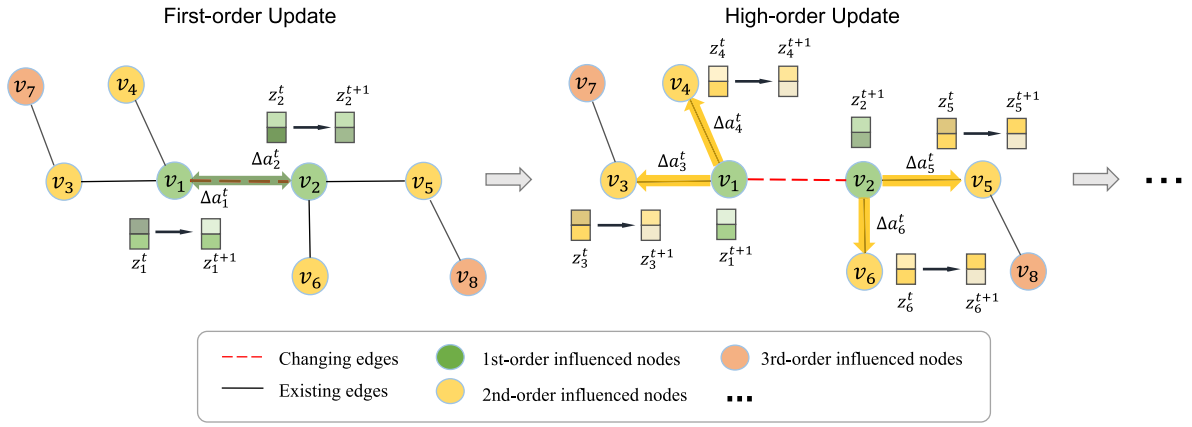

Fig. 2. Overview of DyGCN. An edge emerging at time t directly influences nodes v_1 and v_2, which are first updated by the first-order propagation. The updated v_1 and v_2 then affect their neighbors v_3, v_4, v_5, and v_6, which are updated accordingly to the changed aggregated message. Then, the nodes further away are updated in turn.

Formally, for a 1st-order influenced node $v \in V_1^t$, the change in its aggregated information at time t can be calculated as[1]

$$\Delta a_v^t = \sum_{u \in \mathcal{N}^{t+1}(v) \cup v} z_u^t - \sum_{u \in \mathcal{N}^t(v) \cup v} z_u^t \tag{11}$$

where z_v^t is the embedding of node v at time step t. $\mathcal{N}^t(v)$ and $\mathcal{N}^{t+1}(v)$ denote the neighbors of node v at time t and $t+1$, respectively. It is actually the summation of the embeddings of newly appearing neighbors that subtracts that of the disappearing neighbors. For example, v_1 in Fig. 2 has a new neighbors v_2, the change in its aggregated message is thus the embedding of node v_2, i.e., $\Delta a_1^t = z_2$. If the edge between v_1 and v_2 is removed instead, then v_1 loses the neighbor v_2, in which case the change in its aggregated message $\Delta a_1^t = -z_2$.

The original graph convolution generates the node embedding based on the aggregated message through a transformation matrix. Drawing a lesson from this, an extra transformation matrix $W_1 \in \mathbb{R}^{d \times d}$ is introduced to model the influence of the change in the aggregated message on the embeddings of the 1st-order influenced nodes. Specifically, updating the node embedding based on the original node embedding and the calculated change in the aggregated message is a neural-network-based function, that is,

$$z_v^{t+1} = \sigma\left(W_0 z_v^t + W_1 \Delta a_v^t\right) \tag{12}$$

where $W_0 \in \mathbb{R}^{d \times d}$ is a transformation matrix for the original embeddings.

2) High-Order Update: Because the 1st-order influenced nodes have been updated, their changed embeddings are propagated to their neighbors and in turn influence their embeddings. Here, the update of the 2nd-order influenced nodes is introduced, which can be easily generalized to the neighbors further away.

More specifically, the aggregated message of the 2nd-order influenced nodes is altered because the embeddings of some neighbors (the 1st-order influenced nodes) have changed. Different from the 1st-order influenced nodes influenced by the change in the topological structure, the 2nd-order influenced nodes are influenced by the change in the embeddings of their neighbors. Formally, we can calculate the change in the aggregated message of a 2nd-order influenced node $v \in V_2^t$ which can be calculated as

$$\Delta a_v^t = \sum_{u \in \mathcal{N}^{t+1}(v) \cup v} \left(z_u^{t+1} - z_u^t\right). \tag{13}$$

In essence, it is only the change in the 1st-order influenced nodes in their neighborhood since the others' embeddings remain unchanged. In the example in Fig. 2, the node v_3 aggregates the embeddings of its neighbor, the 1st-order influenced node v_1, whose embedding has been updated to z_1^{t+1}. Accordingly, the change in its aggregated message is $z_1^{t+1} - z_1^t$.

A similar update function to that in (12) is used to update the embeddings of a 2nd-order influenced node v with another transformation matrix $W_2 \in \mathbb{R}^{d \times d}$

$$z_v^{t+1} = \sigma\left(W_0 z_v^t + W_2 \Delta a_v^t\right). \tag{14}$$

The update of the 2nd-order influenced nodes is propagated iteratively to their neighbors, which can be updated in the same way. Formally, for the kth-order influenced nodes $v \in \mathcal{V}_k^t$, they aggregate changed message from the (k-1)th-order influenced nodes as in 13 and update itself

$$z_v^{t+1} = \sigma\left(W_0 z_v^t + W_k \Delta a_v^t\right) \tag{15}$$

where $W_k \in \mathbb{R}^{d \times d}$ is the transformation matrix for the kth propagation layer. Overall, the algorithm of DyGCN can be summarized as Algorithm 1.

3) Processing of New Nodes: In contrast to the existing methods [24], [32], the proposed method is capable of handling the change (removal or addition) in nodes and edges in a unified way rather than separately. Fig. 1 illustrates an example. The new nodes emerging at time t can be considered as isolated nodes with no neighbors before time t. In this case, there is a need only to define a large enough adjacency matrix A^0 to ensure that its size is equal to or higher than the maximum number of nodes through all the time steps. For each new node at time step t, the embeddings are initialized as the aggregation of its linked nodes at time step t. Therefore,

[1]For notional clarity, we use z to denote the node embedding in dynamic setting, distinguished from h in the section of "GCN."

Authorized licensed use limited to: INSTITUTE OF AUTOMATION CAS. Downloaded on November 03,2022 at 06:50:37 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Algorithm 1 DyGCN

Require: $\mathcal{G}^t = \{\mathcal{V}^t, \mathcal{E}^t\}, G^{t+1} = \{\mathcal{V}^{t+1}, \mathcal{E}^{t+1}\}$: the graph at times t and $t+1$;
$Z_t = \{z_v^t, v \in \mathcal{V}^t\}$: the node embeddings at time t;
$\{W_0, W_1, W_2, \ldots, W_K\}$: the transformation matrices.
Ensure: $Z^{t+1} = \{z_v^{t+1}, v \in \mathcal{V}^t\}$, the node embeddings at time $t+1$.

// The update of first-order influenced nodes
for $v \in \mathcal{V}_1^t$ **do**
  $\Delta a_v^t = \sum_{u \in \mathcal{N}^{t+1}(v) \cup v} z_u^t - \sum_{u \in \mathcal{N}^t(v) \cup v} z_u^t$;
  $z_v^{t+1} = \sigma(W_0 z_v^t + W_1 \Delta a_v^t)$;
end for
// The update of high-order influenced nodes
for $k \in [2, \ldots, K]$ **do**
  for $v \in \mathcal{V}_k^t$ **do**
    $\Delta a_v^t = \sum_{u \in \mathcal{N}^{t+1}(v) \cup v} (z_u^{t+1} - z_u^t)$;
    $z_v^{t+1} = \sigma(W_0 z_v^t + W_k \Delta a_v^t)$.
  end for
end for

the addition or removal of nodes can be considered as that of edges and can be processed in a unified way.

4) Matrix Form: The updating process is described in detail from the perspective of the above embedding propagation scheme. To provide a holistic view of the embedding propagation and to facilitate batch implementation, the matrix form of the update rule as in GCN [18] is given here, denoted as a dynamic graph convolution. An extra subscript is introduced to denote the layer index of dynamic graph convolution in DyGCN. The above discussed 1st-order update can be summarized in the following expression, called the 1st-order dynamic graph convolution:

$$\begin{aligned} Z_1^{t+1} &= \text{DYGCONV}_1\left(\Delta A^t, Z^t, W_0, W_1\right) \\ &:= \sigma\left(\Delta \hat{A}^t Z^t W_1 + Z^t W_0\right) \end{aligned} \tag{16}$$

where Z_1^{t+1} denotes the node embeddings after the 1st-order update. Because this work aims only to update the 1st-order influenced nodes, all the other nodes are masked in the above updating process.

For the k-order influenced nodes ($k > 2$), their update can be written as (17), called the k-order dynamic graph convolution:

$$\begin{aligned} Z_k^{t+1} &= \text{DYGCONV}_k\left(A^{t+1}, \Delta Z^t, Z^t, W_0, W_k\right) \\ &:= \sigma\left(\hat{A}^{t+1} \Delta Z^t W_k + Z^t W_0\right) \end{aligned} \tag{17}$$

where $\Delta Z^t = Z_{k-1}^{t+1} - Z_{k-2}^{t+1}$. Z_k^{t+1} denotes the node embeddings after the k-order is updated. Similarly, all the nodes are masked in the above updating process, except the k-order influenced nodes. After K times of dynamic graph convolution, the final node embeddings Z^{t+1} are obtained by updating all the k-order nodes with the corresponding column of Z_k^{t+1}.

5) Unsupervised Training: To learn meaningful and predictive representations in a fully unsupervised setting, a structure-preserving loss function is applied to the updated node embeddings to tune the transformation matrices $\{W_0, W_1, \ldots, W_K\}$. The training dataset is generated as a sequence of graphs. We use a trained static graph embedding method to get Z^0 and reconstruct the graph at other times by a structure-preserving loss function as the training process. The structure-preserving loss function encourages the nearby nodes to have similar embeddings, while enforcing that the embeddings of disparate nodes are highly distinct [13]

$$L = -\sum_t \sum_v \left(\log\left(\sigma\left(z_v^\top z_u\right)\right) + \log\left(1 - \sigma\left(z_v^\top z_{u^-}\right)\right)\right) \tag{18}$$

where u is a neighbor of v, u^- is a random node, and σ is the sigmoid function. The unsupervised loss aims to preserve the topological information, which requires no extra supervision. It can be replaced, or augmented, by a task-specific objective (e.g., cross-entropy loss) for specific downstream tasks.

C. Spectral DyGCN

In this section, Spectral DyGCN is introduced to further improve the above-mentioned high-order update mechanism with acceptable time consumption.

The high-order update mechanism described above ignores the nodes with orders higher than k. This means that the changing information of the 1st-order nodes cannot be propagated to all the nodes, which is reasonable because the nodes that are far from the "center" of change are less likely to be affected. However, this inevitably results in loss of accuracy. To further emphasize accuracy, a method that considers the global propagation of a graph is introduced here. However, the method is difficult to carry out in the spatial form of GCN given that spatial convolution initially considers each node separately. Therefore, the graph propagation step in the spectral domain is considered.

In the graph theory, the normalized graph Laplacian is defined as $L = I_n - D^{-(1/2)} A D^{-(1/2)}$, where I_n is the identity matrix. The normalized Laplacian can be decomposed as $L = U \Lambda U^{-1}$, where $\Lambda = \text{diag}([\lambda_1, \ldots, \lambda_n])$ denotes its eigenvalues. U is the $n \times n$ square matrix whose ith column is the eigenvector u_i. The graph Fourier transform of a signal x is defined as $\hat{z} = U^{-1} z$, whereas the inverse transform is $x = U\hat{x}$. Then the original network propagation $D^{-(1/2)} A D^{-(1/2)} z$ can be interpreted as x is first transformed into the spectral space and scaled by the eigenvalues, and then transformed back

$$z' = D^{-\frac{1}{2}} A D^{-\frac{1}{2}} z = U(I_n - \Lambda) U^{-1} z. \tag{19}$$

It has been proven that the scale of eigenvalues controls the propagation process partially or modularly [51]. For better approximation of the dynamic propagation, the network propagation of (19) is generalized into a learnable form

$$z' = U g_\theta(\Lambda) U^{-1} z. \tag{20}$$

As proven in [18], (20) can be well-approximated by a truncated expansion in terms of the Chebyshev polynomials. Thus, the graph propagation step can be formulated as

$$Z' \approx W_s \left(I_N + D^{-\frac{1}{2}} A D^{-\frac{1}{2}}\right) Z \tag{21}$$

where W_s is the parameter matrix after approximation. Equation (21) is the exact formulation of the vanilla GCN

Authorized licensed use limited to: INSTITUTE OF AUTOMATION CAS. Downloaded on November 03,2022 at 06:50:37 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Algorithm 2 Spectral DyGCN

Require: $\mathcal{G}^t = \{\mathcal{V}^t, \mathcal{E}^t\}, \mathcal{G}^{t+1} = \{\mathcal{V}^{t+1}, \mathcal{E}^{t+1}\}$: the graph at times t and $t+1$;
$Z_t = \{z_v^t, v \in \mathcal{V}^t\}$: the node embeddings at time t;
$\{W_0, W_1, W_2, \ldots, W_K\}$: the transformation matrices.
Ensure: $Z^{t+1} = \{z_v^{t+1}, v \in \mathcal{V}^t\}$, the node embeddings at time $t+1$.
// The update of first-order influenced nodes
for $v \in \mathcal{V}_1^t$ **do**
$\Delta a_v^t = \sum_{u \in \mathcal{N}^{t+1}(v) \cup v} z_u^t - \sum_{u \in \mathcal{N}^t(v) \cup v} z_u^t$;
$z_v^{t+1} = \sigma(W_0 z_v^t + W_1 \Delta a_v^t)$
end for
// Propagating the changes to the whole graph
$Z^{t+1} = W_s(I_N + D^{-\frac{1}{2}} A D^{-\frac{1}{2}}) Z^{t+1}$

TABLE I
TIME-CONSUMING ANALYSIS ON DIFFERENT COMPARED METHODS. $|\mathcal{E}^t|$ AND $|\mathcal{V}^t|$ DENOTE THE NUMBER OF EDGES AND NODES OF THE GRAPH, RESPECTIVELY. $|\Delta\mathcal{E}^t|$ AND $|\Delta\mathcal{V}^t|$ DENOTE THE NUMBER OF CHANGING EDGES AND NODES, RESPECTIVELY. d IS THE DIMENSIONALITY OF NODE EMBEDDINGS. r, l, AND w ARE PARAMETERS OF THE RANDOM WALK-BASED METHODS, I.E., THE NUMBER OF WALKS, THE WALK LENGTH, AND THE WINDOW SIZE CORRESPONDINGLY

Methods	Global updating	Re-training
GCN	$O(d\|\mathcal{E}^t\|)$	✓
DNE	$O(d\|\Delta\mathcal{E}^t\|)$	✓
DynWalks	$O(\|\Delta\mathcal{E}^t\| + \|\mathcal{V}^t\| + rwl\|\Delta\mathcal{V}^t\|)$	✓
GraphSAGE	$O(d\|\Delta\mathcal{V}^t\|)$	✗
DyGCN	$O(d(\|\Delta\mathcal{E}^t\| + \|\Delta\mathcal{V}^t\|))$	✗
Spectral DyGCN	$O(d(\|\Delta\mathcal{E}^t\| + \|\mathcal{V}^t\|))$	✗

model in the spectral domain. In this case, Spectral DyGCN changes mainly the high-order update part of DyGCN to propagate the changing information to the whole graph; the algorithm is expressed as Algorithm 2.

D. Model Analysis

In this section, DyGCN is compared with other models in terms of effectiveness and efficiency. Section V presents the observed results. There are two main aspects of time consumption problems in common dynamic graph embedding: global update consumption and retraining time consumption. Table I presents a summary of the comparison of different methods.

1) DyGCN Versus Static Methods (e.g., GCN): The vanilla GCN designed for the static setting consists of multiple layers of graph convolution, which update the node embeddings between two consecutive layers. In contrast, the proposed DyGCN is designed for the dynamic setting, which consists of multiple layers of dynamic graph convolution to update the node embeddings between two consecutive time steps. The 1st-order and high-order updates can be approximately seen as graph convolutions on the change in adjacency matrix ΔA^t and node embeddings ΔZ^t, respectively. The time complexity of graph convolution comes mainly from the matrix multiplication of the adjacency matrix, node embedding matrix, and transformation matrix. Satisfactorily, there is mostly 0 in ΔA^t, and the number of nonzero entries in ΔA^t is equal to the number of the 1st-order influenced nodes. Similarly, the number of nonzero entries in ΔZ^t at the k-order update is equal to the number of k-order influenced nodes. Compared with static GCN, which updates the embeddings of all the nodes, what is updated in DyGCN $V_1^t \cup \cdots \cup V_K^t$ is a subset of the whole node set $\mathcal{V}^t$. Considering K is usually small (2 by default), the number of updated nodes is much less than the size of the whole node set; thus, considerable running time is saved. In addition, the vanilla GCN is designed to learn the parameters of each graph for a better convolution process, which means that the model needs to be trained at every time step. Thus, the training process is very time costly.

2) DyGCN Versus DNE and DynWalks: DNE [7] and DynWalks [15] are the dynamic extended versions of LINE and DeepWalk, respectively. Both consider how to update only a few nodes according to the changes, which means that they mainly solve the global update consumption problem. However, these two methods update the node embeddings by stochastic gradient descent; thus, the optimization progress during update is necessary. In contrast, DyGCN does not require optimization progress once the model has been trained, making it much more time-saving.

3) DyGCN Versus GraphSAGE: In contrast to the above-mentioned vanilla GCN, GraphSAGE also solves the efficiency issues but at the cost of performance. When new nodes emerge or the node embeddings are changed, GraphSAGE simply recalculates the embedding of each influenced node individually and simultaneously. However, each node is influenced by its neighboring nodes, and its change will influence its neighbors in turn. GraphSAGE is unable to model this progressive influence among nodes, nor can it capture the evolving pattern, which makes its performance even worse. In conclusion, although GraphSAGE works quicker than DyGCN, it fails to capture the influence of the change in neighboring nodes and the evolving pattern on the graph. Hence, its performance in dynamic graph embedding is considered worse than that of DyGCN.

4) Spectral DyGCN Versus Other Methods: Spectral DyGCN captures the dynamic propagation mechanism and does not need to retrain any parameters during the node update, thus decreasing the time consumption considerably. In contrast to DyGCN, it updates the whole graph at each snapshot. Consequently, Spectral DyGCN is more powerful but slightly more costly than DyGCN.

V. EXPERIMENTS

In this section, the experiments are described in detail, including the experiment setup, the comparison of the effectiveness and efficiency of different models, the analysis of the update order, and visualizations.

A. Experiment Setup

1) Datasets: In accordance with [7], [11], experiments were carried out on the following three real-world datasets, the statistics of which are shown in Table II.

Authorized licensed use limited to: INSTITUTE OF AUTOMATION CAS. Downloaded on November 03,2022 at 06:50:37 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

TABLE II
STATISTICS OF THE THREE EVALUATION DATASETS (*-* DENOTES THE NUMBER OF CHANGES ACROSS DIFFERENT SNAPSHOTS)

Data set	No. of nodes	No. of edges	No. of time steps
AS	7716	8809-26467	60
HEP-TH	8557-14351	23318-47776	60
Facebook	2085-2235	9092-88642	36

1) *Autonomous Systems (AS)*[2] [21] is a communication network from the border gateway protocol logs. The dataset contains instances spanning from 1997 to 2000. Sixty consecutive snapshots in the dataset were selected for evaluation.
2) *HEP-TH*[3] [8] is a dataset containing abstracts of reports in High Energy Physics Theory conferences from 1993 to 2003. A collaboration network was created for reports published in each month. Sixty consecutive snapshots in the dataset were selected.
3) *Facebook*[4,5] [40] is a social network dataset on the Facebook website. A subset of Amherst College was selected. To generate a sequence of dynamic networks, a subgraph from the original network was randomly sampled as the initial networks. At each time step t, a fixed number of new nodes and edges was added.

To evaluate the ability of the models to predict future changes on graphs based on past changes, the first half of time steps was used as the training set and the second half of time steps as the test set for the three datasets. Note that the dynamic graphs have a certain order. The use of the earlier graphs for validating the later graphs for training may lead to the risk of data leakage. The last graph snapshot of the training sets was used for validation. The reported results were averaged over all the snapshots in the test set.

2) Compared Methods: The proposed DyGCN model was compared with the following methods:

1) *GraphSAGE* [13] is an inductive GCN-based method that can effectively learn the representation of new nodes but cannot model the evolving pattern through the time steps on dynamic graphs.
2) *TIMERS* [52] is a dynamic embedding model based on incremental SVD with a theoretically instructed maximum error-bounded restart.
3) *DNE* [7] is an extension of the skip-gram-based methods to the dynamic setting, with high efficiency.
4) *DynWalks* [15] is an extension of DeepWalk [31], which can dynamically and efficiently learn embeddings based on those selected, with high efficiency.
5) *GCN* [18], [19] is a prevalent static embedding model and the static counterpart of the proposed DyGCN model. In dynamic scenarios, each snapshot of the dynamic graphs is regarded as a static graph, and GCN is used to learn its node embeddings. The node degrees are used as node features.

[2]https://snap.stanford.edu/data/as-733.html
[3]https://snap.stanford.edu/data/cit-HepTh.html
[4]http://people.maths.ox.ac.uk/porterm/data/facebook100.zip
[5]https://github.com/lundu28/DynamicNetworkEmbedding/tree/master/data

It is important to note that GCN is used as the static base model to obtain the original embeddings at each time step in DyGCN and to learn the embeddings at the next time steps. We show the performance of GCN here aiming to give the theoretical upper bound of performance of all the dynamic graph embedding methods. However, the dynamic methods update the embeddings based on the historical graph embeddings. The updating process inevitably results in loss of information. The dimension of node embeddings d is 100 for all the methods. The maximum order of update K is set as 2 by default.

3) Tasks and Evaluation Metrics: Experiments were carried out, and the methods were compared on their performance of the following two tasks:

1) *Link Prediction*: This aims to test how well the learned embeddings can predict unobserved edges. The edge proximity is measured based on the cosine distance between the embeddings of pairs of nodes. To achieve this aim, the models are evaluated on their ability to predict future links at $t + 1$ time step using embeddings learned at t time step. The area under the ROC curve (**AUC**) and **F1-score** are used as the evaluation metrics for this task. As discussed in II-B, DyGCN is not designed for link prediction. However, link prediction is a traditional task in evaluating the quality of the learned node embedding.
2) *Node Classification*: To test the classification ability of the embeddings, logistic regression is used as the classifier, with the node embeddings as inputs. At the initial time step, Z^0 is used as the training data for the logistic regression parameters, which are kept unchanged in the following time step. **Accuracy** is the evaluation metric of node classification at each time step. We summarize the predicted result of all the nodes at each time using the updating embeddings. The logistic regression uses L-BFGS optimization with l2 regularization; the tolerance for stopping criteria is 0.0001.

B. Results and Analysis

Table III shows the link prediction performance and running time (seconds) of the evaluation at each time step for the three datasets. Fig. 3 shows the node classification performance for the Facebook dataset. Note that GCN, which achieved the best performance and consumed the longest running time, is the base model of DyGCN. Its performance was the upper bound of DyGCN in theory.

Considering both Table III and Fig. 3, GraphSAGE showed the worst performance in all the cases. This result was expected because the method was originally designed for static embedding and failed to capture the changing patterns during the time steps, as well as the progressive influence from its neighbors. However, GraphSAGE was the fastest among the methods due to incremental computation; i.e., each new node needed only to aggregate information from its neighbors for representation; its neighboring nodes would not be affected. TIMERS achieved a moderate performance with a little running time. It updated only a part of the nodes at each

Authorized licensed use limited to: INSTITUTE OF AUTOMATION CAS. Downloaded on November 03,2022 at 06:50:37 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

TABLE III

COMPARISON OF THE LINK PREDICTION PERFORMANCE AND RUNNING TIME (SECONDS) OF DIFFERENT METHODS. THE BEST PERFORMANCE AMONG DYNAMIC EMBEDDING METHODS IS HIGHLIGHTED. GCN IS A STATIC METHOD THAT PERFORMS THE UPPER BOUND IN THEORY

	AS			HEP-TH			Facebook		
	AUC	F1	TIME	AUC	F1	TIME	AUC	F1	TIME
GraphSAGE	0.5130	0.5421	0.2149	0.5576	0.5347	0.4134	0.5178	0.5151	0.0289
TIMERS	0.6763	0.6699	0.2235	0.5497	0.5326	0.2681	0.7228	0.5646	0.4337
DNE	0.5818	0.6149	92.0739	0.6551	0.6185	111.3373	0.7395	0.6733	68.7416
DynWalks	0.8541	0.7674	13.5219	0.8867	0.8229	17.5355	0.7488	0.7231	8.1941
DyGCN	0.8621	0.7694	0.3138	0.8964	0.8679	0.5973	0.7544	0.7430	0.0350
Spectral DyGCN	**0.8738** (±0.0124)	**0.7710** (±0.0171)	0.4374	**0.9039** (±0.0103)	**0.8806** (±0.0168)	0.7549	**0.7697** (±0.0054)	**0.7482** (±0.0097)	0.0548
GCN	0.8935	0.7731	221.8825	0.9324	0.8983	453.9417	0.8184	0.7493	143.9285

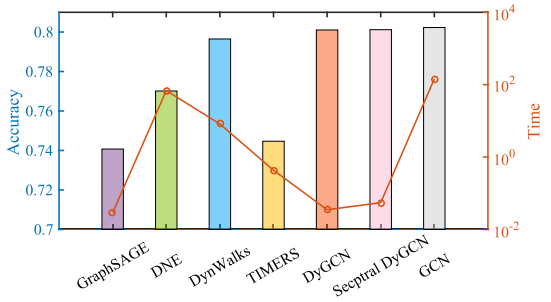

Fig. 3. Performance comparison of the node classification task on the Facebook dataset along with running time (seconds).

time step and did not require any optimization progress during updating. The performance was moderate because TIMERS is based on the matrix perturbation theory, which works only when the adjacency matrix does not change considerably at every time step [52]. Both DNE and DynWalks need SGD for updating the nodes; thus, they consumed much more time than the other methods. DNE achieved a better performance than TIMERS for the HEP-TH and Facebook datasets but at the cost of much higher time consumption. For the AS dataset, DNE was not obviously better than TIMERS.

From our perspective, the graph on the AS dataset was much sparser than the other datasets, and an adding edge would not significantly affect the adjacency matrix; thus, TIMERS obtained comparatively better results for AS. DyWalks achieved a better performance than DNE, being mainly motivated by incremental skip-gram, which is well-designed for efficiency and effectiveness. The method updated only the nodes that could be reached by random walk with the start of the 1st-order nodes. Note that GCN is a transductive graph embedding method, while GraphSAGE is an inductive graph embedding method. When the topological structure of the graph changes, the parameters should be relearned. Therefore, it is unable to learn graph embeddings efficiently in dynamic setting. However, GraphSAGE is able to calculate the graph embeddings when the graph structure changes, but its performance in dynamic settings cannot be guaranteed.

The proposed DyGCN consistently yielded the best performance in all the cases with a limited time cost, indicating its superiority in both accuracy and efficiency. It achieved a comparative performance with DynWalks, and its running time was the same as that of GraphSAGE and TIMERS. Compared with the base model GCN, DyGCN achieved a competitive performance in both link prediction and node classification with a much shorter running time. Quantitatively, DyGCN was 692×, 759×, and 4110× faster than its static counterpart GCN for the AS, HEP-TH, and Facebook datasets, respectively.

This was reasonable because the number of updated nodes was much smaller than the number of all the nodes, and there was no need for a training process during the update. Spectral DyGCN performed better than DyGCN and had an acceptable running time compared with the other methods. This made sense given that Spectral DyGCN updated the whole graph instead of only a part of nodes, as in DyGCN. Note that GCN, which achieved the best performance and had the longest running time, is the base model of DyGCN. Its performance was the upper bound of DyGCN in theory. In summary, the experimental results verified the effectiveness and efficiency of the proposed DyGCN, which achieved the most comparable results to GCN with much less time consumption.

C. Long-Term Dynamic Embedding

To test the robustness of the proposed model in case of cumulative errors in long-term embedding, the models were compared on their performance of the long-term tasks of link prediction and node classification. In the long-term setting of these two tasks, only the initial node embeddings at $t = 0$ were given. The calculated node embeddings at $t - 1$ were used as the initial node embeddings at time step t. Thus, the node embeddings at each time step t varying from 1 to 30 were learned, given only the node embedding at $t = 0$.

Fig. 4 shows the results at every step, which indicate that GraphSAGE achieved a comparatively worst performance among all the methods because it did not capture any evolving pattern of dynamic changes. TIMERS achieved high accuracy for the AS dataset because, as discussed above, the method is suitable for a sparser graph, such as that of the AS dataset. TIMERS, GraphSAGE, and DyGCN, which do not require optimization progress, were stabler through all the time steps compared with the other methods. This may be due to the unstable randomness caused by the optimization progress. The performance of all the methods clearly decreased with the increase in time steps. However, it is interesting to note that the performance tended to have an unpredictable vibration over 20 $\sim$ 30 time steps for both the AS and Facebook datasets, whereas there is not too big vibration in HEP-TH. The present authors believe that the sparser the graph, the harder it is to represent each node

Authorized licensed use limited to: INSTITUTE OF AUTOMATION CAS. Downloaded on November 03,2022 at 06:50:37 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

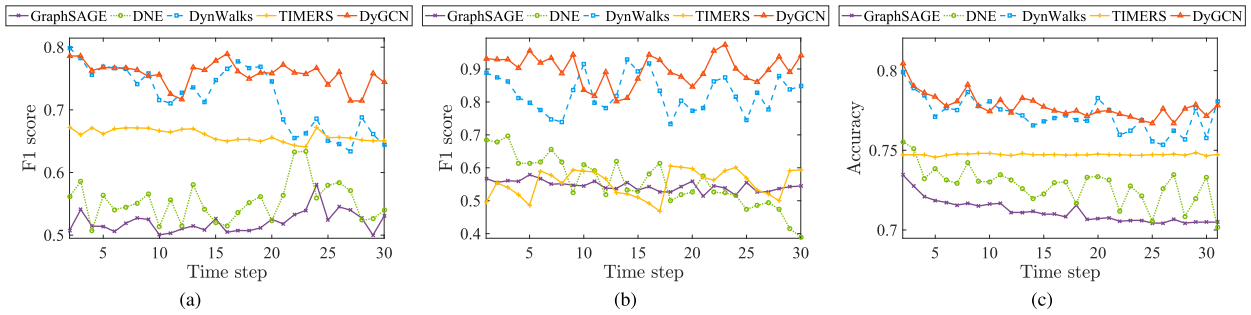

Fig. 4. Performance comparison of the long-term link prediction task w.r.t. F1-score on AS dataset, HEP-TH dataset, and node classification task on the Facebook dataset. (a) AS. (b) HEP-TH. (c) Facebook.

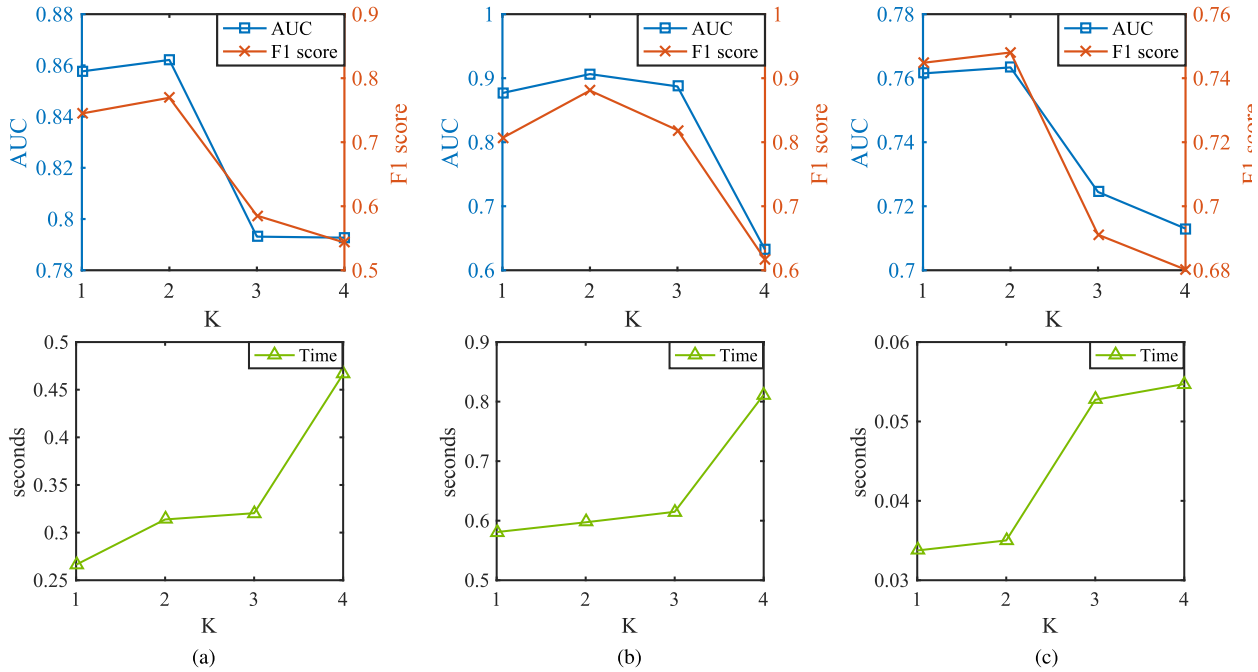

Fig. 5. Performance and running time w.r.t. different orders of update K. (a) AS. (b) HEP-TH. (c) Facebook.

accurately. The proposed DyGCN and DynWalks consistently outperformed the other methods for all the datasets. From the holistic perspective, DyGCN performed better than DynWalks, especially when it was a long time away from initialization. Moreover, the performance of DynWalks was observed to decrease sharply for the AS dataset, whereas that of DyGCN was much stabler. Overall, conclusion can be drawn that DyGCN has a robust performance in long-term tasks, which is very important in real-world applications.

D. Update Order

The maximum updated order K was varied in the range of $\{1, 2, 3, 4\}$ to determine how it would affect the model accuracy and efficiency. Fig. 5 shows a summary of the experimental results; the top image presents the findings for DyGCN, and the bottom image shows the time consumption. With the increase in update order, the performance first increased and then decreased sharply after $K > 2$ for the AS and Facebook datasets. This suggests that the change mostly affects the 1st- and 2nd-order influenced nodes, whereas the influence on nodes further away is relatively negligible. Propagating the change to further nodes and updating them is superfluous and can deteriorate the performance. However, both the AUC and the F1-score were relatively high up to $K = 3$ for HEP-TH, whose graph was much denser and edge changes much fewer compared with the other datasets. The reason for this may be that HEP-TH had much less 1st-order nodes and much more 2nd- and (especially) 3rd-order nodes than the other datasets. Thus, when DyGCN is applied to a graph with less 1st-order and more 3rd-order nodes, a bigger update order may be necessary. In addition, the increase in K resulted in higher time consumption, especially after $K > 3$. This is reasonable because the number of influenced nodes increases exponentially with the increase in the update order. When $K > 3$, the time consumption was even higher than that of Spectral DyGCN. Therefore, it is better to update only the 1st- and 2nd-order influenced nodes, for the sake of both effectiveness and efficiency.

E. Dimensionality of Hidden Vectors

The models were tested with embedding dimensionality of latent vector $d = 50, 75, 100$, and 125. As shown in Fig. 6,

Authorized licensed use limited to: INSTITUTE OF AUTOMATION CAS. Downloaded on November 03,2022 at 06:50:37 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

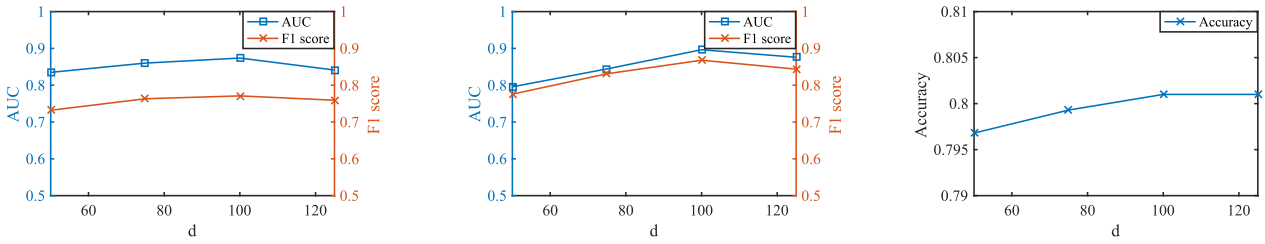

Fig. 6. Performance of DyGCN of different hidden vector sizes d in the three datasets. (a) AS. (b) HEP-TH. (c) Facebook.

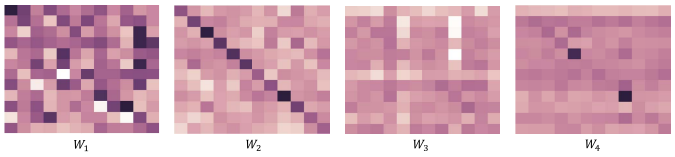

Fig. 7. Visualization of graph kernel matrix W_k of DyGCN in the AS dataset.

DyGCN had a relatively stable performance with varying dimensionalities for all the datasets. When the embedding dimensionality was high, DyGCN tended to be overfitting. DyGCN achieved the best performance with $d = 100$ for all the three datasets.

F. Graph Convolution Kernel Visualization

To better understand the function of the graph convolution operators, the weight of graph convolution kernels W_k was visualized. For the AS dataset for example, as shown in Fig. 7, W_1 is the transform matrix between the nodes that have an additional link at this time step, whereas W_2, W_3, and W_4 are the transform matrices of the higher order update. W_1 and W_2 have obvious activated parts, whereas most parts in W_3 and W_4 are not well-activated. This suggests that W_1 and W_2 capture a clear pattern during optimization, whereas W_3 and W_4 seem to have difficulty in learning a distinct node propagation pattern. This may be another explanation of why DyGCN obtained the best performance with only the 1st- and 2nd-order updates. Besides, W_2 has high weights in the diagonal line, which means that the nodes tend to pay more attention to their original information than the neighborhood information during their 2nd-order propagation. This makes sense because the influence of neighborhood information should be smaller than that of the 1st-order update.

VI. Conclusion

This work focused on how to extend GCN models to learning node embeddings efficiently in a dynamic graph setting. The proposed DyGCN naturally generalizes the embedding propagation scheme of GCN to the dynamic setting in an efficient manner, by propagating the change along the graph to update the node embeddings. The most affected nodes are updated first, and then their changes are propagated to further nodes, which in turn are updated. Two different methods of updating are proposed: high-order update and spectral propagation. The experimental results show the efficiency and effectiveness of DyGCN and Spectral DyGCN compared with other methods and their static counterpart (GCN).

References

[1] A. T. Balaban, "Applications of graph theory in chemistry," *J. Chem. Inf. Comput. Sci.*, vol. 25, no. 3, pp. 334–343, 1985.
[2] D. Beck, G. Haffari, and T. Cohn, "Graph-to-sequence learning using gated graph neural networks," 2018, *arXiv:1806.09835*.
[3] M. Belkin and P. Niyogi, "Laplacian Eigenmaps and spectral techniques for embedding and clustering," in *Proc. NIPS*, 2002, pp. 585–591.
[4] S. Cao, W. Lu, and Q. Xu, "GraRep: Learning graph representations with global structural information," in *Proc. 24th ACM Int. Conf. Inf. Knowl. Manag.*, Oct. 2015, pp. 891–900.
[5] J. Chen, T. Ma, and C. Xiao, "FastGCN: Fast learning with graph convolutional networks via importance sampling," 2018, *arXiv:1801.10247*.
[6] Z. Cui, Z. Li, S. Wu, X.-Y. Zhang, and L. Wang, "Dressing as a whole: Outfit compatibility learning based on node-wise graph neural networks," in *Proc. World Wide Web Conf.*, May 2019, pp. 307–317.
[7] L. Du, Y. Wang, G. Song, Z. Lu, and J. Wang, "Dynamic network embedding: An extended approach for skip-gram based network embedding," in *Proc. 27th Int. Joint Conf. Artif. Intell.*, Jul. 2018, pp. 2086–2092.
[8] J. Gehrke, P. Ginsparg, and J. Kleinberg, "Overview of the 2003 KDD cup," *ACM SIGKDD Exp. Newslett.*, vol. 5, no. 2, pp. 149–151, Dec. 2003.
[9] M. Gori, G. Monfardini, and F. Scarselli, "A new model for learning in graph domains," in *Proc. IEEE Int. Joint Conf. Neural Netw.*, May 2005, pp. 729–734.
[10] P. Goyal, S. R. Chhetri, and A. Canedo, "Dyngraph2Vec: Capturing network dynamics using dynamic graph representation learning," *Knowl.-Based Syst.*, vol. 187, Jan. 2020, Art. no. 104816.
[11] P. Goyal, N. Kamra, X. He, and Y. Liu, "DynGEM: Deep embedding method for dynamic graphs," 2018, *arXiv:1805.11273*.
[12] A. Grover and J. Leskovec, "Node2Vec: Scalable feature learning for networks," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2016, pp. 855–864.
[13] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proc. NIPS*, 2017, pp. 1024–1034.
[14] C. T. Have and L. J. Jensen, "Are graph databases ready for bioinformatics?" *Bioinfmatics*, vol. 29, no. 24, p. 3107, 2013.
[15] C. Hou, H. Zhang, K. Tang, and S. He, "DynWalks: Global topology and recent changes awareness dynamic network embedding," 2019, *arXiv:1907.11968*.
[16] Z. Huang, W. Chung, and H. Chen, "A graph model for E-commerce recommender systems," *J. Amer. Soc. Inf. Sci. Technol.*, vol. 55, no. 3, pp. 259–274, 2004.
[17] S. M. Kazemi et al., "Representation learning for dynamic graphs: A survey," *J. Mach. Learn. Res.*, vol. 21, no. 70, pp. 1–73, 2020.
[18] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2016, *arXiv:1609.02907*.
[19] T. N. Kipf and M. Welling, "Variational graph auto-encoders," in *Proc. NIPS Workshop*, 2016, pp. 1–3.
[20] J. B. Kruskal, *Multidimensional Scaling*, vol. 11. Newbury Park, CA, USA: SAGE, 1978.
[21] J. Leskovec and A. Krevl, "SNAP datasets: Stanford large network dataset collection," SNAP Group, Stanford Univ., Stanford, CA, USA, Tech. Rep., 2014. [Online]. Available: http://snap.stanford.edu/data
[22] J. Li, H. Dani, X. Hu, J. Tang, Y. Chang, and H. Liu, "Attributed network embedding for learning in a dynamic environment," in *Proc. ACM Conf. Inf. Knowl. Manag.*, Nov. 2017, pp. 387–396.
[23] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," 2015, *arXiv:1511.05493*.

Authorized licensed use limited to: INSTITUTE OF AUTOMATION CAS. Downloaded on November 03,2022 at 06:50:37 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

[24] J. Li, K. Cheng, L. Wu, and H. Liu, "Streaming link prediction on dynamic attributed networks," in *Proc. 11th ACM Int. Conf. Web Search Data Mining*, Feb. 2018, pp. 369–377.
[25] F. Manessi, A. Rozza, and M. Manzo, "Dynamic graph convolutional networks," *Pattern Recognit.*, vol. 97, Jan. 2020, Art. no. 107000.
[26] K. Marino, R. Salakhutdinov, and A. Gupta, "The more you know: Using knowledge graphs for image classification," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 20–28.
[27] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013, *arXiv:1301.3781*.
[28] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. NIPS*, 2013, pp. 3111–3119.
[29] S. A. Myers, A. Sharma, P. Gupta, and J. Lin, "Information network or social network? The structure of the Twitter follow graph," in *Proc. 23rd Int. Conf. World Wide Web*, 2014, pp. 493–498.
[30] A. Pareja et al., "EvolveGCN: Evolving graph convolutional networks for dynamic graphs," 2019, *arXiv:1902.10191*.
[31] B. Perozzi, R. Al-Rfou, and S. Skiena, "DeepWalk: Online learning of social representations," in *Proc. 20th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2014, pp. 701–710.
[32] B. Paassen, D. Grattarola, D. Zambon, C. Alippi, and B. E. Hammer, "Graph edit networks," in *Proc. ICLR*, 2020, pp. 1–34.
[33] S. T. Roweis and L. K. Saul, "Nonlinear dimensionality reduction by locally linear embedding," *Science*, vol. 290, no. 5500, pp. 2323–2326, Dec. 2000.
[34] A. Sankar, Y. Wu, L. Gou, W. Zhang, and H. Yang, "DySAT: Deep neural representation learning on dynamic graphs via self-attention networks," in *Proc. 13th Int. Conf. Web Search Data Mining*, Jan. 2020, pp. 519–527.
[35] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Trans. Neural Netw.*, vol. 20, no. 1, pp. 61–80, Jan. 2008.
[36] Y. Seo, M. Defferrard, P. Vandergheynst, and X. Bresson, "Structured sequence modeling with graph convolutional recurrent networks," in *Proc. Int. Conf. Neural Inf. Process.*, 2018, pp. 362–373.
[37] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, "LINE: Large-scale information network embedding," in *Proc. 24th Int. Conf. World Wide Web*, May 2015, pp. 1067–1077.
[38] L. Tang, H. Liu, J. Zhang, and Z. Nazeri, "Community evolution in dynamic multi-mode networks," in *Proc. 14th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD)*, 2008, pp. 677–685.
[39] J. B. Tenenbaum, V. de Silva, and J. C. Langford, "A global geometric framework for nonlinear dimensionality reduction," *Science*, vol. 290, no. 5500, pp. 2319–2323, Dec. 2002.
[40] A. L. Traud, P. J. Mucha, and M. A. Porter, "Social structure of Facebook networks," *Phys. A, Stat. Mech. Appl.*, vol. 19, pp. 4165–4180, Feb. 2012.
[41] R. Trivedi, M. Farajtabar, P. Biswal, and H. Zha, "DyRep: Learning representations over dynamic graphs," in *Proc. 7th Int. Conf. Learn. Represent. (ICLR)*, New Orleans, LA, USA, 2019.
[42] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," 2017, *arXiv:1710.10903*.
[43] D. Wang, P. Cui, and W. Zhu, "Structural deep network embedding," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2016, pp. 1225–1234.
[44] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics Intell. Lab. Syst.*, vol. 2, nos. 1–3, pp. 37–52, 1987.
[45] F. Wu, T. Zhang, A. H. de Souza, Jr., C. Fifty, T. Yu, and K. Q. Weinberger, "Simplifying graph convolutional networks," 2019, *arXiv:1902.07153*.
[46] S. Wu, Y. Tang, Y. Zhu, X. Xie, and T. Tan, "Session-based recommendation with graph neural networks," in *Proc. AAAI*, 2018, pp. 1–10.
[47] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," 2019, *arXiv:1901.00596*.
[48] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" 2018, *arXiv:1810.00826*.
[49] B. Yu, H. Yin, and Z. Zhu, "Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting," 2017, *arXiv:1709.04875*.
[50] W. Yu, W. Cheng, C. C. Aggarwal, K. Zhang, H. Chen, and W. Wang, "NetWalk: A flexible deep embedding approach for anomaly detection in dynamic networks," in *Proc. 24th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Jul. 2018, pp. 2672–2681.
[51] J. Zhang, Y. Dong, Y. Wang, J. Tang, and M. Ding, "ProNE: Fast and scalable network representation learning," in *Proc. 28th Int. Joint Conf. Artif. Intell.*, Aug. 2019, pp. 4278–4284.
[52] Z. Zhang, P. Cui, J. Pei, X. Wang, and W. Zhu, "Timers: Error-bounded SVD restart on dynamic networks," in *Proc. 32nd AAAI Conf. Artif. Intell.*, 2018, pp. 1–10.
[53] Z. Jie, G. Cui, Z. Zhang, C. Yang, Z. Liu, and M. Sun, "Graph neural networks: A review of methods and applications," 2018, *arXiv:1812.08434*.
[54] L. Zhou, Y. Yang, X. Ren, F. Wu, and Y. Zhuang, "Dynamic network embedding by modeling triadic closure process," in *Proc. AAAI*, 2018, pp. 1–8.
[55] Y. Zuo, G. Liu, H. Lin, J. Guo, X. Hu, and J. Wu, "Embedding temporal network via neighborhood formation," in *Proc. 24th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Jul. 2018, pp. 2857–2866.

Authorized licensed use limited to: INSTITUTE OF AUTOMATION CAS. Downloaded on November 03,2022 at 06:50:37 UTC from IEEE Xplore. Restrictions apply.