Cache-based GNN System for Dynamic Graphs

Haoyang Li, Lei Chen
The Hong Kong University of Science and Technology
Hong Kong SAR, China
{hlicg,leichen}@cse.ust.hk

ABSTRACT

Graph Neural Networks (GNNs) have achieved great success in downstream applications due to their ability to learn node representations. However, in many applications, graphs are not static. They often evolve with changes, such as the adjustment of node attributes or graph structures. These changes require node representations to be updated accordingly. It is non-trivial to apply current GNNs to update node representations in a scalable manner. Recent research proposes two types of solutions. The first solution, sampling neighbors for the influenced nodes, requires expensive processing for each node. The second solution, reducing the repeated computations by merging the shared neighbors, cannot speed up the updating process if the influenced nodes do not share neighbors. Most importantly, the above solutions ignore the hidden representations obtained in the previous times that can be reused to accelerate the representation updating. In this paper, we propose a general cache-based GNN system to accelerate the representation updating. Specifically, we cache a set of hidden representations obtained in the previous times, and then reuse them in the next time. To identify valuable hidden representations, we first estimate the number of hidden representations and their combinations that can be reused. Secondly, we formulate the k-assembler problem that selects k representations to maximize the saved time for the next updating process. Experiments on three real-world graphs show that the cache-based GNN system can significantly speed up the representation updating for various GNNs.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Computer systems organization** → **Neural networks**.

KEYWORDS

Graph Neural Network; Cache; Dynamic Graph

ACM Reference Format:
Haoyang Li, Lei Chen. 2021. Cache-based GNN System for Dynamic Graphs. In *Proceedings of the 30th ACM International Conference on Information and Knowledge Management (CIKM '21), November 1–5, 2021, Virtual Event, QLD, Australia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3459637.3482237

CIKM '21, November 1–5, 2021, Virtual Event, QLD, Australia

ACM ISBN 978-1-4503-8446-9/21/11…$15.00
https://doi.org/10.1145/3459637.3482237

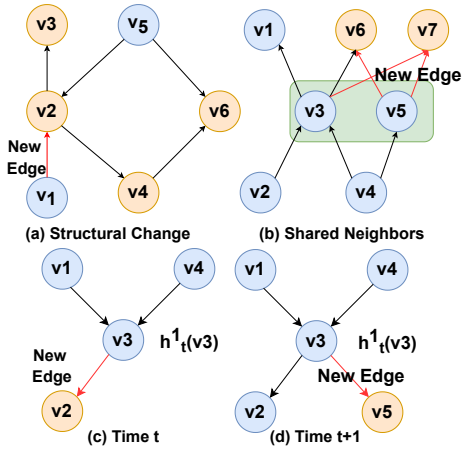


Figure 1: Red edges denote new coming edges, and orange nodes are influenced that their representation need to be updated. $\mathbf{h}_t^1(v_3)$ is the hidden representation at first layer of v_3.

1 INTRODUCTION

Graph Neural Networks (GNNs) have achieved great success in various graph-based applications, such as network analysis [29, 43], community detection [37], recommendation [30, 45], and spam detection [7, 26]. GNNs learn node representations through a layer-wise message passing manner to aggregate the information from neighbors recursively. Specifically, a message-passing layer can be separated into **AGGREGATE** and **COMBINE** operations, i.e., GNNs first aggregate the information from the neighbors of each node and then combine the aggregated result and the node's representation to output the hidden representation in the next layer.

Unlike static graphs, dynamic graphs consist of various graph changes, such as the adjustment of node attributes or graph structures. For example, in social platforms, the users can change their attributes, and join or leave the platforms, and connections among users can appear and disappear. These graph changes affect node representations and these node representations should be updated to capture graph changes accordingly. Taking Figure 1 (a) as an example, the representations of nodes $\{v_2, v_3, v_4, v_6\}$ are influenced by the new edge $e(v_1, v_2)$ and should be updated.

To capture graph changes, researchers [6, 14, 21, 42] apply traditional GNNs to learn node representations on the latest graph snapshot. Furthermore, to incorporate temporal information into node representation, researchers [5, 31, 33, 35] first perform traditional GNN models on each graph snapshot and then consider interactions among different snapshots by recurrent neural network [46] or self-attention mechanism [41]. The above works have achieved great success on the effectiveness of various tasks, such as the precision of node classification. However, due to the high

Table 1: Following [19, 26], we list four GNNs as examples. GCN [21], GIN [44], and GraphSAGE-P [14] are order-invariant models that do not consider the order of neighbors., while GraphSAGE-LSTM [14] as an order-matter GNN considers the order of neighbors. Specifically, GraphSAGE-P and GraphSAGE-LSTM denote the max-pooling and LSTM variants of the GraphSAGE, respectively. σ and max denote non-linear activation and max functions, respectively.

GNN Model	AGGREGATE	COMBINE
GCN	$\mathbf{h}^l(\mathcal{IN}(v)) = \sum_{u \in \mathcal{IN}(v)} \mathbf{h}^{l-1}(u)$	$\mathbf{h}^l(v) = \sigma(W^l \cdot \frac{\mathbf{h}^l(\mathcal{IN}(v)) + \mathbf{h}^{l-1}(v)}{\lvert\mathcal{IN}(v)\rvert+1})$
GIN	$\mathbf{h}^l(\mathcal{IN}(v)) = \sum_{u \in \mathcal{IN}(v)} \mathbf{h}^{l-1}(u)$	$\mathbf{h}^l(v) = \sigma(W \cdot ((1+\epsilon^l) \cdot \mathbf{h}^l(\mathcal{IN}(v)) + \mathbf{h}^{l-1}(v))$
GraphSAGE-P	$\mathbf{h}^l(\mathcal{IN}(v)) = \max_{u \in \mathcal{IN}(v)} \{\sigma(W_1^K \cdot \mathbf{h}^{l-1}(u))\}$	$\mathbf{h}^l(v) = \sigma(W_2^l \cdot (\mathbf{h}^l(\mathcal{IN}(v)), \mathbf{h}^{l-1}(v))$
GraphSAGE-LSTM	$\mathbf{h}^l(\mathcal{IN}(v)) = LSTM(\mathbf{h}_{v_1}^{l-1}, \cdots, \mathbf{h}_{v_k}^{l-1})$	$\mathbf{h}^l(v) = \sigma(W^l \cdot (\mathbf{h}^l(\mathcal{IN}(v)), \mathbf{h}^{l-1}(v))$

computation cost of AGGREGATE operation and the high update frequency of graphs, applying these GNNs to learn node representations efficiently on large dynamic graphs is still a challenge.

Currently, there are mainly two types of research works to address the challenge. The first type of works, including GraphSAGE [14], PinSAGE [45], and FastGCN [4] etc., improve the scalability by first sampling a small set of neighbors for influenced nodes, and then aggregating their information. However, sampling neighbors is expensive since it needs to be processed recursively [19]. The second type of works, including HAG [19] etc., speed up the GNN computations by merging frequently shared neighbors into a supernode, which eliminates the repeated computations. Nevertheless, if influenced nodes do not share any neighbors, they will not accelerate the updating process on the dynamic graph.

Most importantly, current works neglect to reuse hidden representations of nodes obtained in the previous steps to accelerate the representation updating. For example, in Figure 1 (c)(d), when computing the representation of influenced node v_2 at time t, we can obtain the hidden representation $\mathbf{h}_t^1(v_3)$ of the node v_3. Thus, when we update the representation of v_5 at time $t+1$, we can reuse $\mathbf{h}_t^1(v_3)$ instead of re-computing it from scratch, which saves the computation time. In addition to the hidden representations of single nodes, we can pre-compute the aggregated results of frequently shared neighbors, i.e., hidden presentation combinations. They can accelerate the AGGREGATE operation. For example, if we use GraphSAGE-SUM [14] to update node representation of influenced nodes v_6 and v_7, it will sum hidden representation of the neighbors of v_6 and v_7 in AGGREGATE operation. Thus, we can pre-compute the combined representation $\{\mathbf{h}_t^1(v_3) + \mathbf{h}_t^1(v_5)\}$ to accelerate computations.

The above observations motivate us to utilize hidden representations and their combinations to accelerate representation updating. Inspired by caching technique [23, 28], which has been used to speed up various tasks, such as query [3, 24] and access optimization [32], we propose a novel and general cache-based GNN framework. It can be applied to various GNN models [14, 31, 40, 42, 44]. Specifically, before new graph changes coming, we cache hidden representations and their combinations into a limited cache with size k, to speed up the representation learning in next time. In fact, caching hidden representations within a limited size is quite challenging and non-trivial. Firstly, we should infer the saved time of each hidden representation and their combinations in the next time efficiently and effectively. Secondly, since different representations may be overlapped, the total saved time of the cached representations is not equal to the sum of them. For example, in Figure 1 (b), if we have cached $\{\mathbf{h}_t^1(v_3) + \mathbf{h}_t^1(v_5)\}$ to update v_6 and v_7, caching $\mathbf{h}_t^1(v_3)$ and $\mathbf{h}_t^1(v_5)$ cannot get extra benefit. Thus, we should select k representations to maximize the total saved time under the consideration of the overlapping among these representations.

To address the above two challenges, we first analyze various cases where the hidden representations will be utilized in the next time theoretically. Based on the theoretical analysis, we then propose an efficient and effective algorithm to approximate the utilization number of the hidden representations and the combinations of representations, where the utilization number can be used to compute the saved time of each representation. Second, with the utilization number of hidden representations, we first formulate the k-assembler problem that aims at finding k representations from the hidden representations and their combinations to maximize the total saved time in next time. We then prove that the problem is NP-hard, and propose an efficient algorithm to solve it with a theoretical approximation guarantee. We summarize the contributions of this paper as follows.

- We present a novel and general cache-based GNN framework to speed up representation learning of GNNs on dynamic graphs.
- We theoretically present the cases where hidden representations will be utilized, and propose an algorithm to approximate their expected saved time in an efficient and effective manner.
- We formulate the k-assembler problem that selects k hidden representations to maximize the total saved time. We prove that it is NP-hard, and propose an efficient algorithm with approximation ratio guarantee as the solution.
- Extensive experiments on three real-world dynamic graphs show the superior acceleration effect on representation learning.

In the following sections, we first discuss the related work in Section 2 and introduce the important definitions in Section 3. We then describe the details of our proposed predictor in Section 4 and the cache assembler in Section 5. We present our experimental results in Section 6 and conclude in Section 7.

2 RELATED WORK

Graph Neural Network. Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, GNNs follow the layer-wise message passing manner to encode nodes $v \in \mathcal{V}$ into low-dimensional vector space (i.e., $\mathbf{v} \in \mathbb{R}^d$) by recursively aggregating the information from its in-neighbors $u \in \mathcal{IN}(v)$. Formally, the representation of nodes can be achieved by:

$$\mathbf{h}^{l+1}(\mathcal{IN}(v)) = AGGREGATE(\{\mathbf{h}^l(u) | u \in \mathcal{IN}(v)\}), \quad (1)$$

$$\mathbf{h}^{l+1}(v) = COMBINE(\mathbf{h}^{l+1}_{\mathcal{IN}(v)}, \mathbf{h}^l(v)), \quad (2)$$

where $\mathbf{h}^{l+1}(\mathcal{IN}(v))$ and $\mathbf{h}^{l+1}(v)$ denote the aggregated representation of v's in-neighbors and the representation of v at $l+1$ layer,

Table 2: The important notations used in this paper

Notation	Description
$\mathcal{IN}_t^l(v)$, $\mathcal{ON}_t^l(v)$	l hop in and out neighbors of v at time t
$\mathbf{h}_t^l(v)$, $\mathbf{h}_t(v)$	l layer and final representation at time t
$\mathcal{V}_{chi}(\mathbf{h}_t^l(v))$, $\mathcal{V}_{par}(\mathbf{h}_t^l(v))$	The child and parent nodes of $\mathbf{h}_t^l(v)$
$\mathbf{h}_t^r(\mathcal{V}_i)$, $\mathbf{h}_t^r(\mathcal{V}_j)$	The general hidden representation index
$STU(\mathbf{h}_t^l(v))$	Saved time per utilization of $\mathbf{h}_t^l(v)$
$UTN(\mathbf{h}_t^l(v))$	Utilization number of $\mathbf{h}_t^l(v)$
$ST(R)$	Saved time of the representation set R
$\mathcal{V}_{com}(\mathbf{h}_t^r(\mathcal{V}_i))$	Nodes are involved to obtain $\mathbf{h}_t^r(\mathcal{V}_i)$

respectively, AGGREGATE is the function to aggregate its neighbors' representations , and COMBINE is to update the representation of nodes. Note that the AGGREGATE and COMBINE can be customized by different GNNs. We list four representative work in Table 1, i.e., GCN [21], GIN [44], GraphSage-P [14], and GraphSage-LSTM [14]. To improve the efficiency of GNNs, PGE [17], FastGCN [4], ASGCN [18] sample neighbors to decrease aggregated time. We propose another solution from different view that we store the hidden representation in cache to accelerate representation learning. Also, our general cache-based system can be applied on above GNN-based models [4, 14, 17, 18, 21, 44] to accelerate the updating process in the dynamic graphs without loss of task performance.

Temporal Graph Neural Network. This type work incorporates temporal information into node representation. DySAT [35] first learns the representations at each time snapshot and learn their interactions by self-attention mechanism [36]. WD-GCN [31] and GC-LSTM [5] first use GCN to learn node representation at each time snapshot, and use LSTM to capture their temporal features. EvolveGCN [33] integrates a RNN into a GCN to update the weight parameters of the GCN. DynGEM [11] uses a deep autoencoder to encode snapshots of discrete node dynamic graphs. DyREP [39] and LDG [22] learn node representations by posing representation learning as a latent mediation process, bridging the dynamic processes of topological evolution and node interactions.

3 PROBLEM DEFINITION

In this section, we briefly introduce the concepts related to our problem and formally give the problem definition.

Definition 3.1 (Dynamic Graph). We define the dynamic graph from time 1 to t as $\mathcal{DG} = \{\mathcal{G}^1, \cdots \mathcal{G}^t\}$ where $\mathcal{G}^t = \{\mathcal{V}^t, \mathcal{E}^t, \mathbf{A}^t, \mathbf{X}^t\}$ denotes the graph at the time t, and $\mathcal{V}^t$, $\mathcal{E}^t$, $\mathbf{A}^t$, and $\mathbf{X}^t$ denote the nodes, edges, the adjacent matrix, and node attributes, respectively. In particular, $\mathcal{IN}_t^l(v)$ and $\mathcal{ON}_t^l(v)$ denote the l-hop in and out-neighbors of the node v at time t, respectively.

Definition 3.2 (Graph Change). Graph changes include attribute changes and structural changes: (1) Attribute Change: attribute change events are defined as $E_{t+1}^a = \{e_a = \{v, x_v^{t+1}, t+1\}\}$ where e_a indicates node v has an updating attribute x_v^{t+1} at time $t+1$. (2) Structural Change: we use $E_{t+1}^s = \{e_s = \{v, u, t+1, \tau\}\}$ to denote the structural change event from v (i.e., source node) to u (i.e., target node) at time $t+1$. $\tau = 0$ and $\tau = 1$ denote edge deletion and edge addition, respectively.

Definition 3.3 (Influenced Node). Given a GNN $\mathcal{M}$ with L layers, each attribute change event influences its neighbors within L-hop, and each structure event influences the target node's out-neighbors within $L-1$ hop. Formally, given $E_{t+1}^a = \{e_a = \{v, x_v^{t+1}, t+1\}\}$ and $E_{t+1}^s = \{e_s = \{v, u, t+1, \tau\}\}$, and GNN $\mathcal{M}$, the influenced nodes set is $\mathcal{V}_{t+1}^{inf} = \{\mathcal{ON}_{t+1}^{0\sim L}(v)|e_a \in E_{t+1}^a\} \cup \{\mathcal{ON}_{t+1}^{0\sim L-1}(u)|e_s \in E_{t+1}^s\}$.

With graph change events E_{t+1}^a and E_{t+1}^s , we need to update the representation of influenced nodes $\mathcal{V}_{t+1}^{inf}$ accordingly. To accelerate the representation updating, we can cache a set of hidden representations obtained in the previous time. We formally define two types of hidden representations that can be cached as follows:

Definition 3.4 (Representation Candidate). There are two types of representations at time t that can be reused at time $t+1$.

- Single hidden representation: given the GNN $\mathcal{M}$ with L layers, and the dynamic graph $\mathcal{DG}$, single hidden representations of each node $v \in \mathcal{V}^t$ are $\{\mathbf{h}_t^l(v)\}_{l=0}^{L-1}$. We denote the saved time per utilization of hidden representation $\mathbf{h}_t^l(v)$ as $STU(\mathbf{h}_t^l(v))$. $STU(\mathbf{h}_t^l(v))$ is the time of computing $\mathbf{h}_t^l(v)$, which is pre-computed and depends on the specific GNN $\mathcal{M}$.
- Hidden representation combinations: given a set of nodes $\mathcal{V}_i = \{v_j\}_{j=1}^{|\mathcal{V}_i|}$, we define the hidden representation combination of $\mathcal{V}_i$ at the l-th layer as $\mathbf{h}_t^l(\mathcal{V}_i) = AGGREGATE(\{\mathbf{h}_t^l(v)|v \in \mathcal{V}_i\})$. The saved time per utilization of combination is $STU(\mathbf{h}_t^l(\mathcal{V}_i)) = \sum_{v\in\mathcal{V}_i} STU(\mathbf{h}_t^l(v)) + STAGG(\mathbf{h}_t^l(\mathcal{V}_i))$ where $STAGG(\mathbf{h}_t^l(\mathcal{V}_i))$ denotes the aggregation time among $\{\mathbf{h}_t^l(v)|v \in \mathcal{V}_i\}$ by AGGREGATE function. Also, we use $\mathcal{V}_{com}(\mathbf{h}_t^l(\mathcal{V}))$ to denote nodes $\mathcal{V}$ that are involved to obtain $\mathbf{h}_t^l(\mathcal{V})$, i.e., $\mathcal{V}_{com}(\mathbf{h}_t^l(\mathcal{V})) = \mathcal{V}$.

In this paper we use $\mathbf{h}_t^r(\mathcal{V})$ to denote the single representation and their combinations at layer r where $r \in (0, L)$ interchangeably unless it needs to be specific.

We then show the acceleration process with cache. Specifically, given a node v that need to be updated at time $t+1$, the aggregated result at $l+1$ layer can be obtained as follows.

$$\mathbf{h}_{t+1}^{l+1}(\mathcal{IN}_{t+1}^l(v)) = AGG\left(\begin{cases} \mathbf{h}_t^l(\mathcal{V}) & \mathcal{V} \in \mathcal{V}_t^l(v) \\ \mathbf{h}_{t+1}^l(u) & u \in \mathcal{IN}_{t+1}^l(v) \setminus \mathcal{V}_t^l(v) \end{cases}\right),$$

where $\mathcal{V}_t^l(v)$ denotes the neighbors of v whose hidden representation at l layer have been cached. $\mathbf{h}_{t+1}^l(u)$ is not cached and need to be computed from scratch. Thus, we can reuse these cached representation instead of re-computing them to accelerate the representation learning.

Since the cache size k is limited, we need to select the most top-k hidden representation and combinations from all representation candidates to maximize the total saved time. However, it is non-trivial to select these representations. As we mentioned before, due to overlapping utilizations among hidden representations and combinations, the saved time of the cached representations is not equal to the sum of them. Formally, we first define the domination relationship among representations, the overlapping utilization, and then define the total saved time with cached representation.

Definition 3.5 (Domination Relationship). Given two hidden representations $\mathbf{h}_t^r(\mathcal{V}_i)$ and $\mathbf{h}_t^r(\mathcal{V}_j)$, if (1) $\mathbf{h}_t^r(\mathcal{V}_i)$ is a combination of representation and $\mathcal{V}_{com}(\mathbf{h}_t^r(\mathcal{V}_j))$ is a part of $\mathcal{V}_{com}(\mathbf{h}_t^r(\mathcal{V}_i))$, or (2) $\mathbf{h}_t^r(\mathcal{V}_i)$ and $\mathbf{h}_t^r(\mathcal{V}_j)$ are a high level representation and low level representation, respectively, and obtaining $\mathbf{h}_t^r(\mathcal{V}_i)$ needs to aggregate $\mathbf{h}_t^r(\mathcal{V}_j)$, we regard that $\mathbf{h}_t^r(\mathcal{V}_i)$ denominates $\mathbf{h}_t^r(\mathcal{V}_j)$ and denote the

dominated relation as $\mathbf{h}_t^r(\mathcal{V}_i) \rightarrow \mathbf{h}_t^r(\mathcal{V}_j)$. For example, in Figure 1 (b), $\mathbf{h}_t^1(\mathcal{V}) = \{\mathbf{h}_t^1(v_3)+\mathbf{h}_t^1(v_5)\}$ dominates $\mathbf{h}_t^1(v_3)$ and $\mathbf{h}_t^1(v_5)$. Also, $\mathbf{h}_t^2(v_1)$ dominates $\mathbf{h}_t^1(v_3)$ and $\mathbf{h}_t^1(v_5)$.

Definition 3.6 (Overlapping Utilization). For clarification and without loss of generality, we only consider two cases in this paper that the utilizations of representations are overlapped.

(1) $\mathbf{h}_t^r(\mathcal{V}_i)$ dominates $\mathbf{h}_t^r(\mathcal{V}_j)$, i.e., $\mathbf{h}_t^r(\mathcal{V}_i) \rightarrow \mathbf{h}_t^r(\mathcal{V}_j)$, the overlapping utilization $OV(\mathbf{h}_t^r(\mathcal{V}_i), \mathbf{h}_t^r(\mathcal{V}_j)) = UTN(\mathbf{h}_t^r(\mathcal{V}_i))$.
(2) $\mathbf{h}_t^r(\mathcal{V}_i)$ and $\mathbf{h}_t^r(\mathcal{V}_j)$ are in the same level, and $\mathcal{V}_{com}(\mathbf{h}_t^r(\mathcal{V}_i)) \cap \mathcal{V}_{com}(\mathbf{h}_t^r(\mathcal{V}_j)) \neq \emptyset$, and there is no dominated relation between them. Thus, they are overlapped partially. We denote their nearest dominated representation of $\mathbf{h}_t^r(\mathcal{V}_i)$ and $\mathbf{h}_t^r(\mathcal{V}_j)$ as $\mathbf{h}_t^r(\mathcal{V}_k) = AGG(\mathcal{V}_{com}(\mathbf{h}_t^r(\mathcal{V}_i)) \cup \mathcal{V}_{com}(\mathbf{h}_t^r(\mathcal{V}_j)))$, and overlapped utilization is $OV(\mathbf{h}_t^r(\mathcal{V}_i), \mathbf{h}_t^r(\mathcal{V}_j)) = UTN(\mathbf{h}_t^r(\mathcal{V}_k))$.

Example. Given $\mathbf{h}_t^1(\mathcal{V}) = \{\mathbf{h}_t^1(v_1) + \mathbf{h}_t^1(v_2)\}$, $\mathbf{h}_t^1(v_1)$, and $\mathbf{h}_t^1(v_2)$, their maximum utilization number are denoted as $UTN(\mathbf{h}_t^1(\mathcal{V}))$, $UTN(\mathbf{h}_t^1(v_1))$, and $UTN(\mathbf{h}_t^1(v_2))$. Specifically, if we use $\mathbf{h}_t^1(\mathcal{V})$ to accelerate the computation $\{\mathbf{h}_t^1(v_1) + \mathbf{h}_t^1(v_2)\}$, the maximum utilization number of $\mathbf{h}_t^1(v_1)$ and $\mathbf{h}_t^1(v_2)$ will become $UTN(\mathbf{h}_t^1(v_1)) - UTN(\mathbf{h}_t^1(\mathcal{V}))$, and $UTN(\mathbf{h}_t^1(v_2)) - UTN(\mathbf{h}_t^1(\mathcal{V}))$ due to their overlapping utilizations. Also, for two combinations $\mathbf{h}_t^1(\mathcal{V}_i) = \{\mathbf{h}_t^1(v_1)+\mathbf{h}_t^1(v_2)\}$ and $\mathbf{h}_t^1(\mathcal{V}_j) = \{\mathbf{h}_t^1(v_2) + \mathbf{h}_t^1(v_3)\}$, the overlapping happens when computing $\mathbf{h}_t^1(\mathcal{V}_k) = \{\mathbf{h}_t^1(v_1) + \mathbf{h}_t^1(v_2) + \mathbf{h}_t^1(v_3)\}$. Therefore, if we utilize $\mathbf{h}_t^1(\mathcal{V}_i)$ to compute $\mathbf{h}_t^1(\mathcal{V}_k)$, the maximum utilization number of $\mathbf{h}_t^1(\mathcal{V}_j)$ will become $UTN(\mathbf{h}_t^1(\mathcal{V}_j)) - UTN(\mathbf{h}_t^1(\mathcal{V}_k))$.

We also observe the importance of used order of cached representations. Assuming there is node v with m in-neighbors that need to updated at $t+1$, and cached representation set $R_c = \{\mathbf{h}_t^l(\mathcal{V}_i)\}_{i=1}^k$, computing the best saved time of obtaining $h_{t+1}^{l+1}(v)$ need to select disjoint cache nodes set from R_c to achieve the maximum coverage of in-neighbors of v, which is known as Maximum Disjoint Coverage problem [9]. It is NP-hard and cannot be solved optimally in polynomial time unless P=NP. Thus, we use a greedy method to decide the used order and then compute the saved time. Specifically, we first select $\mathbf{h}_t^r(\mathcal{V}_i)$ with the maximum saved time, i.e., $UTN(\mathbf{h}_t^r(\mathcal{V}_i)) \cdot STU(\mathbf{h}_t^r(\mathcal{V}_i))$. We then update the utilization number of the left cached representations, and select the next one. After the termination of iterations, we can get the used order of cached representations and the saved time in terms of this order. Based on this method, we define the saved time as follows:

Definition 3.7 (Saved Time). Given an ordered hidden representation set $R = \{\mathbf{h}_t^r(\mathcal{V}_i)\}_{i=1}^{|R|}$, the saved time is

$$ST(R) = \sum_{\mathbf{h}_t^r(\mathcal{V}_i) \in R} UTN(\mathbf{h}_t^r(\mathcal{V}_i)) \cdot STU(\mathbf{h}_t^r(\mathcal{V}_i)) - \sum_{j=1}^{|R|-1} OV(R[j], R[0:j-1]) \cdot STU(R[j])$$

where $R[j]$ and $R[0:j-1]$ denotes the j-th representation and representations from the 0-th to j-1-th in the set R, respectively. $OV(R[j], R[0:j-1])$ denotes the overlapping utilization between $R[j]$ and $R[0:j-1]$.

We then define the k-assembler problem as follows:

Definition 3.8 (k-assembler Problem). Formally, given the dynamic graph $\mathcal{DG} = \{\mathcal{G}^1, \cdots \mathcal{G}^t\}$, the GNN model $\mathcal{M}$ with L layers, and the limited cache with size k, we select k representations $R_s = \{\mathbf{h}_t^r(\mathcal{V}_i)\}_{i=1}^k$ from the hidden representations and their combinations at time t, and store them in the cache to maximize the saved time $ST(R_s)$ in next time $t+1$.

To address the above defined problem, we need to handle two tasks as follows.

- **Predictor:** Given dynamic graph $\mathcal{DG} = \{\mathcal{G}^1, \cdots \mathcal{G}^t\}$, GNN $\mathcal{M}$ with L layers, and hidden representations $\{\mathbf{h}_t^l(v)\}_{l=0}^{L-1}$ where $v \in \mathcal{V}^t$, the target is to estimate the utilization number of each hidden representation and their combinations that will be used in the time $t+1$ in an efficient and effective manner, and output a candidate representation set $C_R = \{(\mathbf{h}_t^r(\mathcal{V}_i), UTN(\mathbf{h}_t^r(\mathcal{V}_i)))\}$.
- **Cache Assembler:** Given the candidate representation set $C_R = \{(\mathbf{h}_t^r(\mathcal{V}_i), UTN(\mathbf{h}_t^r(\mathcal{V}_i)))\}$, and the cache size k, the target is to select k representations $R_s = \{\mathbf{h}_t^r(\mathcal{V}_i)\}_{i=1}^k$ from C_R to maximize the saved time $ST(R_s)$ in next time.

Figure 2 depicts the framework overview of the cache-based GNN system, which consists of two components, i.e., *Predictor* and *Cache Assembler*. Before the graph change events at time $t+1$ coming, we have selected the representations based on the predictor and cache assembler. Thus, we can directly use them at time $t+1$ to accelerate the representation updating process.

4 PREDICTOR

In this section, we describe how to estimate the utilization number of hidden representations. Specifically, we present the cases where hidden representations can be reused in next time theoretically, and then propose metrics to measure the utilization number of hidden representations. Finally, we propose an algorithm to search the representation combination candidates.

4.1 Theoretical Analysis

We first define child nodes and parent nodes of each hidden representation, and present two cases where the hidden representation can be used in the next time theoretically.

Definition 4.1 (Child Node and Parent Node). If obtaining the representation of the node v_j need the hidden representation $\mathbf{h}_t^l(v_i)$, the node v_j is regarded as a child node of the hidden representation $\mathbf{h}_t^l(v_i)$. Formally, given a hidden representation $\mathbf{h}_t^l(v_i)$ and GNN $\mathcal{M}$ with L layers, the child node set of $\mathbf{h}_t^l(v_i)$ is denoted as $\mathcal{V}_{chi}(\mathbf{h}_t^l(v_i)) = \mathcal{ON}_t^{0\sim L-l}(v_i)$. Also, the node v_j is a parent of $\mathbf{h}_t^l(v_i)$ if obtaining $\mathbf{h}_t^l(v_i)$ need any hidden representation of v_j. Similarly, the parent node set of the hidden representation $\mathbf{h}_t^l(v_i)$ is denoted as $\mathcal{V}_{par}(\mathbf{h}_t^l(v_i)) = \mathcal{IN}_t^{0\sim l}(v_i)$.

If the hidden representation $\mathbf{h}_t^l(v)$ can be reused in the next time, the hidden representation should be valid and not be involved by graph change events directly, i.e., none of the parent nodes $\mathcal{V}_{par}(\mathbf{h}_t^l(v))$ are involved in attribute events or structural events as the target nodes. Based on this condition, there are two cases where the hidden representation $\mathbf{h}_t^l(v_i)$ can be reused to accelerate the representation learning process:

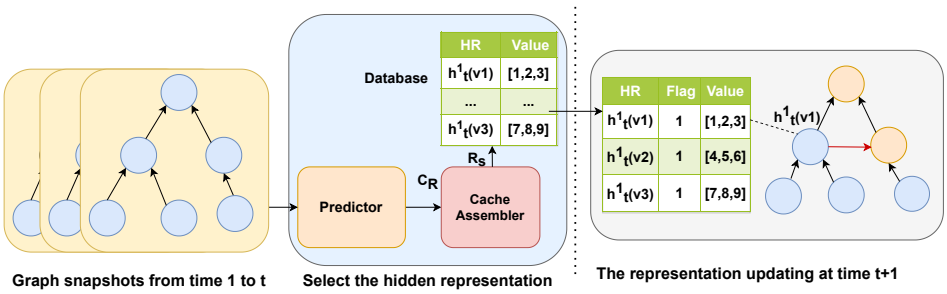


Figure 2: Framework Overview: before time $t+1$, we have selected a set of hidden representations based on the predictor and cache assembler. These cached hidden representations will be used directly to accelerate the updating process at time $t+1$.

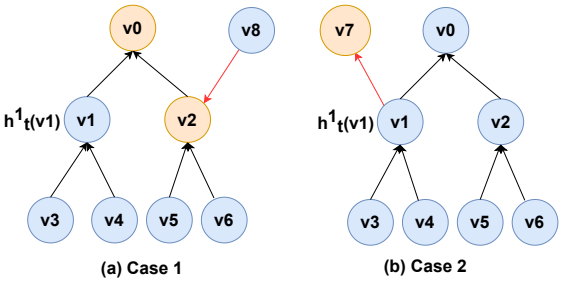


Figure 3: There are two cases that $\mathbf{h}_t^1(v_1)$ can be reused. Layer number of GNN is 2, and orange nodes are influenced nodes.

(1) Case 1: Its child nodes $\mathcal{V}_{chi}(\mathbf{h}_t^l(v_i))$ are influenced. Thus, $\mathbf{h}_t^l(v_i)$ can be reused to obtain their representations. For example, in Figure 3 (a), node v_0 is influenced by new edge $e(v_8, v_2)$ and $\mathbf{h}_t^1(v_1)$ can be reused to obtain the representation of v_0.
(2) Case 2: New nodes are connected as the target nodes by the nodes in $\mathcal{ON}_t^{0\sim L-l-1}(v_i)$. Thus, $\mathbf{h}_t^l(v_i)$ can be used to obtain the representation of these new nodes together with their neighbors. For example, in Figure 3 (b), the node v_7 is connected by v_1. Thus, $\mathbf{h}_t^1(v_1)$ can be used to obtain the representation of v_7.

We then theoretically analyze the utilization number of each $\mathbf{h}_t^l(\mathcal{V}_i)$ at time $t+1$, where $\mathcal{V}_i$ is a node set that can be a single node or a combination of nodes. For case 1, we derive the following Lemma to compute the utilization number of $\mathbf{h}_t^l(\mathcal{V}_i)$.

Lemma 4.2 (Case 1). *Assuming that the total probability that node v will be involved in attribute change events or structural change events as the target node is $p(v)$. The probability $p(\mathbf{h}_t^l(\mathcal{V}_i) \rightarrow v_j)$ that the hidden representation $\mathbf{h}_t^l(\mathcal{V}_i)$ will be used to obtain the latest representation for its child node $v_j \in \mathcal{V}_{chi}(\mathbf{h}_t^l(\mathcal{V}_i))$ is*

$$p(\mathbf{h}_t^l(\mathcal{V}_i) \rightarrow v_j) = (1 - \prod_{v_k \in \mathcal{V}_{tmp}} (1 - p(v_k))) \cdot \prod_{v_k \in \mathcal{V}_{par}(\mathbf{h}_t^l(\mathcal{V}_i))} (1 - p(v_k))$$
$$= \prod_{v_k \in \mathcal{V}_{par}(\mathbf{h}_t^l(\mathcal{V}_i))} (1 - p(v_k)) - \prod_{v_k \in \mathcal{V}_{par}(\mathbf{h}_t(v_j))} (1 - p(v_k)),$$

where $\mathcal{V}_{tmp} = \mathcal{V}_{par}(\mathbf{h}_t(v_j)) \setminus \mathcal{V}_{par}(\mathbf{h}_t^l(\mathcal{V}_i))$. For clarification, we denote $\tilde{p}(\mathbf{h}_t^l(\mathcal{V}_i)) = \prod_{v_k \in \mathcal{V}_{par}(\mathbf{h}_t^l(\mathcal{V}_i))} (1 - p(v_k))$ and $\tilde{p}(\mathbf{h}_t(v_j)) = \prod_{v_k \in \mathcal{V}_{par}(\mathbf{h}_t(v_j))} (1 - p(v_k))$ that indicate the probability of $\mathbf{h}_t^l(\mathcal{V}_i)$ and $\mathbf{h}_t(v_j)$ being not changed at time $t+1$, respectively. Thus, the utilization number of each hidden representation $\mathbf{h}_t^l(\mathcal{V}_i)$ in the case 1 can be computed as:

$$UTN_1(\mathbf{h}_t^l(\mathcal{V}_i)) = \sum_{v_j \in \mathcal{V}_{chi}(\mathbf{h}_t^l(\mathcal{V}_i))} (\tilde{p}(\mathbf{h}_t^l(\mathcal{V}_i)) - \tilde{p}(\mathbf{h}_t(v_j)))$$

Similarly, we compute the utilization number of each hidden representation $\mathbf{h}_t^l(\mathcal{V}_i)$ for case 2 based on the following Lemma.

Lemma 4.3 (Case 2). *Assuming that the node v_j is the l_i-hop out-neighbor of $\mathcal{V}_i$ at time t and will create $n_{v_j}^{new}$ new out-edge with other target nodes at time t+1, denoted as $\mathcal{V}_{t+1}^{tar}(v_j)$. The utilization number of $\mathbf{h}_t^l(\mathcal{V}_i)$ regarding each its l_i-hop out-neighbor v_j can be computed as: $New(v_j, l_i) = \sum_{v_k \in \mathcal{V}_{t+1}^{tar}(v_j)} |\mathcal{ON}_{t+1}^{0\sim L-l-l_i-1}(v_k)|$ where $\mathcal{ON}_{t+1}^{0\sim L-l-l_i}(v_k)$ is out-neighbors of node v_k from the 0 to $L-l-l_i$ hop at time $t+1$, which are influenced and need to be updated accordingly. Thus, the total utilization number of $\mathbf{h}_t^l(\mathcal{V}_i)$ in the case 2 can be computed as:*

$$UTN_2(\mathbf{h}_t^l(\mathcal{V}_i)) = \tilde{p}(\mathbf{h}_t^l(\mathcal{V}_i)) \cdot \sum_{l_i=0}^{L-l-1} \sum_{v_j \in \mathcal{ON}_t^{l_i}(\mathcal{V}_i)} New(v_j, l_i).$$

Therefore, based on the Lemmas 4.2 and 4.3, the expected utilization number of each hidden representation $\mathbf{h}_t^l(\mathcal{V}_i)$ at time $t+1$ is $UTN(\mathbf{h}_t^l(\mathcal{V}_i)) = UTN_1(\mathbf{h}_t^l(\mathcal{V}_i)) + UTN_2(\mathbf{h}_t^l(\mathcal{V}_i))$. In this paper, we regard the edge deletion as a special edge addition by setting its edge from 1 to 0. Thus, all graph change events can fit our systems.

4.2 Utilization Number Predictor

From Lemma 4.2 and 4.3, we can compute the utilization expectation if we know the probability that each hidden representation will be influenced. One direct way is to predict node attribute change events and graph structure change events [8, 34, 48], and then compute these probabilities. However, predicting graph events is heavily time-consuming and complicated. Therefore, for efficiency purpose and simplicity, we compute the probability based on previous graph snapshots, and approximate the utilization number of hidden representations used in next time. Specifically, we define UTN_1 and UTN_2 based on Lemma 4.2 and Lemma 4.3 as follows:

Definition 4.4 (UTN_1). Given the hidden representation $\mathbf{h}_t^l(\mathcal{V}_i)$, we define the utilization number of existing nodes in the case 1 as $UTN_1(\mathbf{h}_t^l(\mathcal{V}_i)) = \sum_{v_j \in \mathcal{V}_{chi}(\mathbf{h}_t^l(v))} (\bar{p}(\mathbf{h}_t^l(\mathcal{V}_i)) - \bar{p}(\mathbf{h}_t(v_j)))$, where $\bar{p}(\mathbf{h}_t(v_j) = \frac{\sum_{t_i=1}^{t-1} \mathbb{I}_{t_i}(\mathbf{h}_{t_i}(v_j))}{t-1}$ and $\bar{p}(\mathbf{h}_t^l(\mathcal{V}_i)) = \frac{\sum_{t_i=1}^{t-1} \mathbb{I}_{t_i}(\mathbf{h}_{t_i}^l(\mathcal{V}_i))}{t-1}$, where the indicator $\mathbb{I}_{t_i}(\mathbf{h}_{t_i}(v_j))$ and $\mathbb{I}_{t_i}(\mathbf{h}_{t_i}^l(\mathcal{V}_i))$ are 1 if the $\mathbf{h}_{t_i}(v_j)$ and $\mathbf{h}_{t_i}^l(\mathcal{V}_i)$ are not changed at time t_i+1, respectively.

Definition 4.5 (UTN_2). Given the hidden representation $\mathbf{h}_t^l(\mathcal{V}_i)$, we define the utilization number in the case 2 as $UTN_2(\mathbf{h}_t^l(\mathcal{V}_i)) = \bar{p}(\mathbf{h}_t^l(\mathcal{V}_i)) \cdot \sum_{l_i=0}^{L-l-1} \sum_{v_j \in \mathcal{ON}_t^{l_i}(\mathcal{V}_i)} \bar{\mathcal{V}}_{t+1}^{tar}(v_j) \cdot \bar{New}(v_j, l_i)$, where $\bar{\mathcal{V}}_{t+1}^{tar}(v_j) = \frac{\sum_{t_i=1}^{t} |\mathcal{V}_{t_i}^{tar}(v_j)|}{t} = \frac{\mathcal{ON}_t^1(v_j)}{t}$ denotes the average number of nodes that v_j connect at time $t+1$, and $\bar{New}(v_j, l_i) = \frac{\sum_{v_k \in \mathcal{ON}_t^{l_i}(v_j)} |\mathcal{ON}_t^{0 \sim L-l-l_i}(v_k)|}{|\mathcal{ON}_t^{l_i}(v_j)|}$ denotes the average utilization number of out-neighbors of v_j that need to updated.

However, it poses another new difficulty in the selection of representation combinations. Due to the exponential search space of combinations, there is no algorithm in polynomial time to compute the utilization number of all combinations. Therefore, to address this challenge, we replace the exponential search space with a smaller candidate search space. One direction is to prune the search space by the rules similar to frequent itemsets mining [2, 12]. But pruning still cannot avoid the explosion of exponential search space, i.e., the notorious pattern explosion phenomenon [13, 47]. Another direction is to choose the combinations whose utilization number is larger than a threshold. Nevertheless, it is #P-hard to select these combinations.

Theorem 4.6. *The problem of counting all combinations whose utilization number is larger than a threshold is #P-hard.*

Proof. We prove it from the reduction of counting the number of τ-support itemsets problem where $\tau \in [0, 1]$ and it is #P-hard [13]. Given m transactions $\mathcal{T} = \{T_1, \cdots, T_m\}$ where each transaction $T_i = \{t_j\}_{j=1}^{|T_i|}$ consists of items, and there are n unique items, i.e., $|\bigcup_{i=1}^{m} T_i| = n$. We build a graph as follows: we regard the n items as nodes at time t and m transactions as new nodes at $t+1$. For each $t_j \in T_i$, we add a directed edge from t_j to T_i. Thus, if we can get all combinations of nodes whose shared new nodes is larger than $\tau \cdot m$, we can obtain a solution for the τ-support itemsets problem. □

Therefore, either directly using the pruning rule or finding the combinations whose utilization number is larger than a threshold are infeasible. The graph property proposes a solution to this dilemma. Due to graph sparsity property, combinations with more nodes tend to be shared by fewer nodes. Moreover, representation combinations with more nodes tend to be unstable because they are more likely to be influenced than a single node. Thus, we believe that combinations with more nodes are not so valuable to explore. Thus, for the efficiency purpose and without losing generality, similar to [19], we only explore 2-combinations.

We also observe that utilization number $UTN(\mathbf{h}_t^r(\mathcal{V}_j))$ is proportional to the number of the 1-hop neighbors of $\mathcal{V}_i$ if we assume the nodes have the same number of neighbors. Therefore, we can design a prune rule to detect the unvaluable combinations and save the time of computing their utilization number. Finally, we can obtain candidate combinations from all 2-combinations and then calculate their exact utilization number.

Prune Rule. Given $\mathcal{V}_c = \{v_i\}_{i=1}^{2}$ with 2 nodes, if their shared out-neighbors satisfies $|\mathcal{ON}_t^1(\mathcal{V}_c)| \leq \frac{min(|\mathcal{ON}_t^1(v_i)||v_i \in \mathcal{V}_c)}{2}$, we will not explore their representation combinations from 0 to $L-1$ layer.

Algorithm 1: Utilization Number Prediction

Input: Dynamic graph $\mathcal{DG} = \{\mathcal{G}^1, \cdots \mathcal{G}^t\}$, the GNN layer number L
Output: The utilization number of the candidate hidden representations $C_R = \{(\mathbf{h}_t^r(\mathcal{V}_i), UTN(\mathbf{h}_t^r(\mathcal{V}_i)))\}$

1 $C_R \leftarrow \emptyset$
2 **foreach** $v \in \mathcal{V}^t$ **do**
3 **for** $l \in [0, L-1]$ **do**
4 $UTN(\mathbf{h}_t^l(v)) \leftarrow$ Definition 4.4 and 4.5
5 $C_R \leftarrow C_R \cup \{(\mathbf{h}_t^l(v), UTN(\mathbf{h}_t^l(v)))\}$
6 $\mathcal{V}_p \leftarrow$ **Obtain_Combination**$(\mathcal{G}^t)$; // By Prune Rule
7 **foreach** $\mathcal{V}_i \in \mathcal{V}_p$ **do**
8 **for** $l \in [0, L-1]$ **do**
9 $UTN(\mathbf{h}_t^r(\mathcal{V}_i)) \leftarrow$ Definition 4.4 and 4.5
10 $C_R \leftarrow C_R \cup \{\mathbf{h}_t^r(\mathcal{V}_i), UTN(\mathbf{h}_t^r(\mathcal{V}_i)))\}$
11 **Return** C_R

Example. If node v and u has three out-neighbors but only share one out-neighbor, i.e., $\mathcal{ON}_t^1(v) = 3, \mathcal{ON}_t^1(u) = 3$, and $\mathcal{ON}_t^1(\{v, u\}) = 1$, we will not consider representation combination of $\{v, u\}$. Due to overlapping relation, it is more worth to select v or u than $\{v, u\}$.

Algorithm 1 shows the detail of computing the utilization number. Given dynamic graph $\mathcal{G}^t = \{\mathcal{V}^t, \mathcal{E}^t\}$, and GNN layer number L, we output a representation set $C_R = \{(\mathbf{h}_t^r(\mathcal{V}_i), UTN(\mathbf{h}_t^r(\mathcal{V}_i)))\}$. We first compute the utilization number of each hidden representation by scanning the L-hop neighbors of each node (lines 2-5). Second, we obtain the potential combination set $\mathcal{V}_p = \{\mathcal{V}_i\}_{i=1}^{|\mathcal{V}_p|}$ by the prune rule, where $\mathcal{V}_i$ denotes the node set (line 6). We then compute the utilization score for these combinations from the 0 to *L-1* layer (lines 7-10). Finally, after the termination of iterations, we obtain all the candidate hidden representations and their utilization number, i.e., $C_R = \{(\mathbf{h}_t^r(\mathcal{V}_i), UTN(\mathbf{h}_t^r(\mathcal{V}_i)))\}$.

Time Complexity. computing utilization number for single hidden representation needs to visit neighbors recursively, which takes $O(L|\mathcal{E}^t| + L|\mathcal{V}^t|)$ (lines 2-5). Also, potential candidate combination selections need $O(d^2|\mathcal{V}^t|)$ time where $d \ll |\mathcal{V}^t|$ is the average number of the out-neighbor of nodes. Similarly, the computation for combinations based on the results of single representation need $O(dL|\mathcal{V}_p|)$ time (lines 7-10). Thus, the total time complexity is $O(L|\mathcal{E}^t| + L|\mathcal{V}^t| + d^2|\mathcal{V}^t| + dL|\mathcal{V}_p|)$.

5 CACHE ASSEMBLER

We prove that the k-assembler problem is NP-hard, and propose an efficient algorithm with a theoretical guarantee to solve it. Finally, we give an effectiveness guarantee that our cache-based GNN system can keep the task performance of the origin GNN model.

5.1 Cache Assembler Algorithm

Theorem 5.1 (Hardness of the k-assembler problem). *The k-assembler problem is NP-hard.*

Proof. We prove that the k-assembler problem is NP-hard by the reduction from the hitting set problem (HSP) [1]. The HSP is defined as follows: given a set of elements $X = \{x_i\}_{i=1}^{|X|}$, a collection of sets $S = \{S_1, S_2, \ldots, S_m\}$, where each subset $S_i = \{x_i\}_{i=1}^{|S_i|}$ consists a set of elements, the target is to select the smallest subset H from X, such that H hits every set comprised in S with the size $|H| \leq k$.

Given an instance $\mathcal{I}$ of the HSP as follows: given a set of elements $X = \{x_i\}_{i=1}^{|X|}$, and the collections of a set of $S = \{S_1, S_2, \ldots, S_m\}$, the target is to find H where $|H| \leq k$ elements that the every set are hit. We transform it into an instance $\mathcal{J}$ of the k-assembler problem by first constructing an element containment set for each element, for example, $Con(x_i) = \{S_j\}_{j=1}^{|Con(x_i)|}$, where S_j contains the element x_i. Thus, the $Con(x_i)$ can be regarded as the hidden representation in the k-assembler problem, and S_j can be regarded as the nodes that can reuse the hidden representation $Con(x_i)$. Also, we assume each $Con(x_i)$ has the same saved time per utilization, and if $Con(x_i)$ and $Con(x_j)$ share common sets, we can regard $Con(x_i)$ and $Con(x_j)$ have a overlapping in the same level. Thus, if we can obtain the optimal results of k-assembler problem, we can union the containment set of each element to decide whether every set is hit in HSP. Thus, we can solve the HSP if we can solve the k-assembler problem. Since the HSP is known to be NP-hard [1], the k-assembler problem is also NP-hard, which completes our proof. □

Theorem 5.1 shows that the k-assembler problem is unlikely to be solved in any polynomial time unless P=NP. Thus, we propose an efficient algorithm to address this problem.

Definition 5.2 (Marginal Gain). Given a set of hidden representations $R_s = \{\mathbf{h}_t^r(\mathcal{V}_i)\}_{i=1}^{|R_s|}$ and a hidden representation $\mathbf{h}_t^r(\mathcal{V}_i)$, we define the conditional marginal gain as follows:

$$\triangle ST(\mathbf{h}_t^r(\mathcal{V}_i)|R_s) = ST(R_s \cup \{\mathbf{h}_t^r(\mathcal{V}_i)\}) - ST(R_s) \tag{3}$$

The basic idea is to select representation with the maximum marginal gain from an empty set until we obtain k hidden representations. In this procedure, the selection order is the same as the used order defined before. To select hidden representations efficiently, we utilize a priority queue Q to maintain each hidden representation with marginal gain and a flag to denote whether the marginal gain is the latest. Thus, in each selection, we update the marginal gain according to Q instead of updating all hidden representations.

The detail is illustrated in Algorithm 2. Given the hidden representation and the cache size k, the target is to obtain the representation set R_s. We first initialize $R_s = \emptyset$ and $Q = \emptyset$, and then compute the marginal gain of each hidden representation and store them into the queue Q (line 1-4). In each iteration, we first pop the element in Q, and find whether its marginal gain is the latest. If it is the latest, we will put it into R_s since the marginal gain of the other representation in Q cannot be larger than $\triangle ST(\mathbf{h}_t^r(\mathcal{V}_i)|R_s)$ (lines 6-8). Thus, we will not check other representations, which saves computation. If the marginal gain is not the latest, we will update the marginal gain and put it into the Q, and reorder the Q (line 10-12). After the termination of the cache assembler algorithm, we will obtain R_s.

We then discuss how good the results are. First, we show that the saved time function is monotone and submodular.

Theorem 5.3. *The saved time function $ST(R)$ in Definition 3.7 is monotone and submodular where R denotes a representation set.*

Proof. The readers can refer to our technique report [27] for full details. □

Algorithm 2: Cache Assembler Algorithm

Input: The hidden representation $C_R = \{\mathbf{h}_t^r(\mathcal{V}_i), UTN(\mathbf{h}_t^r(\mathcal{V}_i))\}_{i=1}^{|C_R|}$, the cache size k
Output: The selected representation set $R_s = \{\mathbf{h}_t^r(\mathcal{V}_i)\}_{i=1}^{k}$

1 $R_s \leftarrow \emptyset$, Queue $Q \leftarrow \emptyset$
2 **foreach** $\mathbf{h}_t^r(\mathcal{V}_i) \in C_R$ **do**
3 $\triangle ST(\mathbf{h}_t^r(\mathcal{V}_i)|R_s) = STU(\mathbf{h}_t^r(\mathcal{V}_i)) * UTN(\mathbf{h}_t^r(\mathcal{V}_i)), flag = 0$
4 Insert $\{\mathbf{h}_t^r(\mathcal{V}_i), flag, \triangle ST(\mathbf{h}_t^r(\mathcal{V}_i)|R_s)\}$ into Q
5 **while** $|R_s| < k$ **do**
6 $\{\mathbf{h}_t^r(\mathcal{V}_i), flag, \triangle ST(\mathbf{h}_t^r(\mathcal{V}_i)|R_s)\} \leftarrow$ Pop first element in Q
7 **if** $flag == |R_s|$ **then**
8 $R_s = R_s \cup \{\mathbf{h}_t^r(\mathcal{V}_i)\}$
9 **else**
10 $\triangle ST(\mathbf{h}_t^r(\mathcal{V}_i)|R_s) \leftarrow$ Equation 3
11 $flag = |R_s|$
12 Insert $\{\mathbf{h}_t^r(\mathcal{V}_i), flag, \triangle ST(\mathbf{h}_t^r(\mathcal{V}_i)|R_s)\}$ into Q and reoder Q
13 **Return** R_s

Since $ST(R)$ is monotone and submodular, according to [16], we have the below theorem.

Theorem 5.4. *Algorithm 2 get an approximation ratio of* $1 - \frac{1}{e}$.

Proof. The readers can refer to our technique report [27] for full details. □

5.2 Effectiveness Guarantee

We guarantee that the effectiveness of GNNs under our cache-based system is the same as the origin GNN model.

Theorem 5.5 (Effectiveness Guarantee). *Given a fixed GNN model $\mathcal{M}$ with fixed parameter $\mathbf{W}$ and a graph $\mathcal{DG}$, the task performance outputted by the origin GNN $\mathcal{M}$ is the same as the task performance outputted by the GNN $\mathcal{M}$ embedded on our cache system.*

Proof. Typically, if representations of the nodes are the same, the performance of the task is the same. Since the cached intermediate hidden representations are aggregated from neighbors, directly utilizing these intermediate results is the same as aggregating them from scratch. Thus, node representations outputted by $\mathcal{M}$ with our cache system are the same as the origin GNN $\mathcal{M}$. □

6 EXPERIMENTS

In this section, we present the experiments on the acceleration effectiveness of our cache-based GNN system for dynamic graphs. We first present the experimental setting and then discuss the results.

6.1 Experimental Setting

6.1.1 Datasets. We briefly introduce three datasets used in our experiments and the detailed statistics of them are listed in Table 3.

UCI [25]. UCI is a dynamic graph where nodes denote users from an online community and edges denotes message communication. The time associated with each edge is their communication time.

HEP [10]. HEP is a dynamic citation graph where nodes denote papers and edges denote citation relationship among papers.

Table 3: Summary statistics of three datasets.

	#nodes	#temporal edges	#time
UCI	**1899**	**59835**	**15**
HEP	**27770**	**352807**	**15**
DBLP	**28085**	**236894**	**15**

DBLP [49]. DBLP is a dynamic co-author network obtained from DBLP, where the nodes denote authors and the edges denote co-authored relationships among authors.

Following the same practice as in [22, 35, 39], we split dynamic datasets into graph snapshots to evaluate our system. Specifically, we split them into 15 snapshots. In each snapshot, we only keep each graph change event once if it occurs many times.

6.1.2 Comparison methods . We use four GNNs to evaluate the accelerate effect of our system: **GCN** [21], **GIN** [44], **GraphSAGE-LSTM**(GS-LSTM) [14], and **WD-GCN** [31]. Specifically, GCN and GIN are order-invariant GNN models that do not consider the order of neighbors, while GS-LSTM is an order-matter GNN that considers the order of neighbors. Moreover, WD-GCN is a representative work of temporal GNNs that it first learn the node representations at each snapshot by GCN, and then feed these representation sequences to LSTM to generate the final node representations. Also, we evaluate our cache-based GNN system with four variants. (1) **Cache**: we store k representations into the cache selected by our proposed algorithm; (2) **Random**: we randomly select k representations from the single representations and their combinations. (3) **Degree**: we sort the hidden representations by the number of their child nodes, and we then select the top-k representations; (4) **Raw**: we do not store any representation in the cache.

6.1.3 Parameter Setting. In this paper, we set the cache size $k = \alpha \cdot |\mathcal{V}^t|$ for different datasets where $\mathcal{V}^t$ denotes node number and α is the parameter to control the cache size k. As default, we set the layer number of GNNs as 2, the representation dimension as 200, $\alpha = 0.6$, and the maximum neighbors of each node as 20. Our system is implemented by Python3.6 and PyTorch and is evaluated on a GPU server with two NVIDIA Tesla P100 GPU(16G) and six CPU core (2.6 GHz) in the Ubuntu 18.04.1 LTS.

We conduct experiments on three datasets and measure the total updating time. We first train GNNs on the first graph snapshot based on the link prediction task. Second, we fix the trainable parameters and evaluate the updating time from the second graph snapshot. Note that before graph events at time $t+1$ coming, we have selected hidden representations and store them in the cache, and thus we only evaluate the updating time in this paper.

Moreover, we do not evaluate the task performance, such as node classification and link prediction, because Theorem 5.5 already shows the task performance of our cache-based GNN model is the same as the original GNN. Thus, the task performance experiments are redundant. More comparisons on task performance of GNNs on the dynamic graphs can be referred to [15, 20, 31, 38].

6.1.4 Implementation Detail. The designed cache stores the selected representations, their index, and valid flags. Thus, at time $t+1$, before we aggregate the representation for influenced nodes, we first access the cache to find the representations that can be reused. Also, different from single hidden representations, the combinations involve multi-nodes. Thus, same as [19], we create a supernode for each selected combination in advance, and build an index to denote it. Thus, we can access its index to reuse it.

Table 4: The updating time(s) and speed-up ratio(×) comparisons between four different GNNs that are supported by different cache selection algorithms on three datasets.

Data	*Model*	*Raw*	*Random*	*Degree*	*Cache*
UCI	***GCN***	4.16	3.26 (1.28×)	2.96 (1.41×)	**1.99(2.09×)**
	GIN	3.71	3.15 (1.18×)	2.72 (1.36×)	**1.82(2.04×)**
	GS-LSTM	63.22	51.96 (1.22×)	47.17 (1.34×)	**29.13(2.17×)**
	WD-GCN	4.78	3.90 (1.23×)	3.73 (1.28×)	**2.53(1.89×)**
HEP	***GCN***	34.26	30.02 (1.14×)	28.67 (1.19×)	**10.22(3.35×)**
	GIN	31.58	29.16 (1.08 ×)	25.26 (1.25×)	**8.37(3.77×)**
	GS-LSTM	806.01	628.90 (1.28×)	591.25 (1.36×)	**202.87(3.97×)**
	WD-GCN	58.62	53.51 (1.10×)	51.83 (1.13×)	**36.87(1.59×)**
DBLP	***GCN***	20.43	18.23 (1.12×)	16.49 (1.24×)	**8.72(2.34×)**
	GIN	19.01	15.75 (1.21×)	14.6 (1.30×)	**7.31(2.60×)**
	GS-LSTM	409.89	330.49 (1.24×)	271.71 (1.51×)	**150.69(2.72×)**
	WD-GCN	39.83	37.75 (1.06×)	35.21 (1.13×)	**29.25(1.36×)**

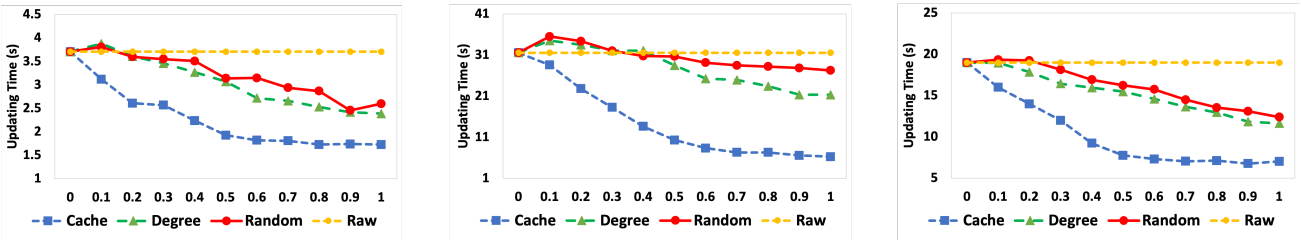
6.2 Experimental Study Results

We report the experimental results on updating time, different cache size, different layer number, and the improvement of combinations.

6.2.1 Evaluation on Updating Time. Table 4 shows that our cache method significantly improves the efficiency of origin GNN model. Also, our cache-based GNN system outperforms Random and Degree on the three datasets, and Random get the worst improvement. Degree can get a better improvement than Random, because the hidden representation with more child nodes tends to be reused in the next time. Nevertheless, Degree ignores the stability of hidden representations and does not consider the new connections with other nodes, and thus its performance is worse than our method. Moreover, the speed-up ratio of WD-GCN is lower than the other three GNN models. It is because after obtaining node representation in the latest timestamp, WD-GCN utilizes the LSTM to encode it with the previous node representations, which costs time.

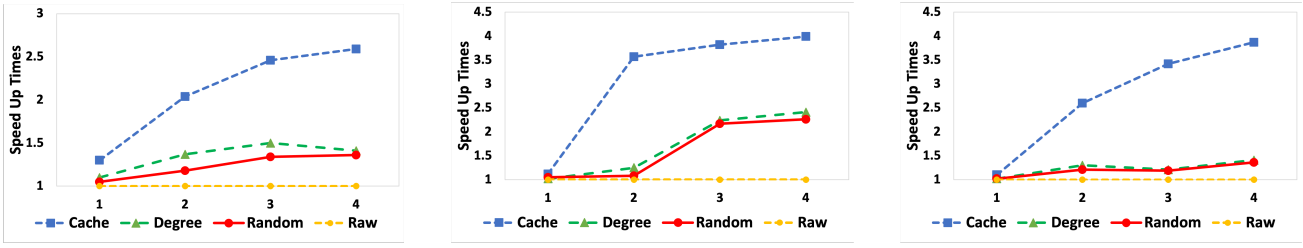
Moreover, from the view of different datasets, Cache gets the best improvement on HEP and gets the worst improvement on the UCI. It is because most of the nodes in HEP stay stable, and each graph change event influences a limited number of nodes. Thus, we can cache a set of stable representations to accelerate updating process. On the other hand, the density of UCI is higher, and the graph change events within each graph snapshot influence a high proportion of nodes in the graph. Thus, the representations stored in the cache are invalid and cannot be used. As a consequence, the speed-up ratio on UCI is lower than the other two datasets.

6.2.2 Evaluation on Different Cache Size. Figure 4 shows that the performance in terms of different cache size k. Specifically, we use GIN as an example. Note that we set $k = \alpha \cdot |\mathcal{V}^t|$, and thus we can control the cache size by the percentage number α. We observe that as the cache size becomes large, the total updating time of the cache method first decreases and then becomes stable. It shows that caching hidden representations can accelerate the updating, but caching more representations will not get extra performance benefit. The reason is that most of the representations cached in the larger cache size cannot be reused or are unstable. Thus, it is unnecessary



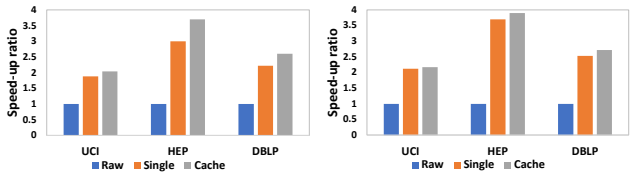
(a) UCI (b) HEP (c) DBLP

Figure 4: Parameter sensitivity analysis in terms of the cache size



(a) UCI (b) HEP (c) DBLP

Figure 5: Parameter sensitivity analysis in terms of the layer number



(a) GIN (b) GraphSage-LSTM

Figure 6: The effect of combinations

to store all hidden representations and their combinations obtained in the previous times, which decreases the waste of storage.

The results also show that the total updating time on Random and Degree decreases continuously, which indicates identifying the most valuable representation can save the cache memory and get the near best improvements on the updating process. Moreover, Random and Degree spend more time than the Raw when $\alpha = 0.1$ on the HEP dataset. It is because the stored representations are too useless to be reused, and accessing the cache costs time.

6.2.3 Evaluation on Different Layer Number. Figure 5 shows the speed-up ratio evaluations in terms of different layers based on GIN on three datasets. We observe that as the layer number increase, Random and Degree increase slowly, while Cache increases significantly. Different from Cache, Random method cannot predict the utilization number of hidden representations. Thus they cannot guarantee the selected representations will be used the next time. Also, Degree only focuses on the number of nodes that need the hidden representation. Thus, as the layer number increases, the hidden representation in the lower layer with less saved time tends to be selected since they can be used for more nodes.

6.2.4 Evaluation on the effect of Combinations. Figure 6 shows the effect of combinations. We evaluate one neighbor order-invariant and one order-matter GNN model, i.e., GIN and GraphSage-LSTM. Specifically, Single only cache single representations. We observe that exploring the combinations can accelerate the speed-up ratio compared with only using single representations. Also, the improvement on GIN is larger than GraphSage-LSTM. It is because GraphSage-LSTM is the order-matter model, and they need to consider the time order of nodes in the combinations. Thus, the number of available combinations to accelerate GraphSage-LSTM decreases.

Summary. Our proposed cache-based GNN system can accelerate representation learning significantly on the three real-world dynamic graphs compared with the baselines. As cache size becomes large, the updating time first decreases and then becomes stable. Also, as the layer number of the GNN model increases, the speed-up ratio increases under the same cache size. Moreover, caching combinations accelerates representation updating.

7 CONCLUSION

This paper proposes a general cache-based GNN system that caches hidden representations obtained in the previous times to accelerate the representation updating in the dynamic graphs with a limited cache size k. Specifically, we first estimate the utilization number of hidden representations and their combinations that will be used in next time, and then select k representations to maximize the total saved time. Extensive experiments on three real-world datasets show that the cache-based GNN system can employ various GNNs and accelerate representation updating significantly.

ACKNOWLEDGEMENT

This work is partially supported by National Key Research and Development Program of China Grant No. 2018AAA0101100, the Hong Kong RGC GRF Project 16202218, RIF Project R6020-19, AOE Project AoE/E-603/18, Theme-based project TRS T41-603/20R, China NSFC No. 61729201, Guangdong Basic and Applied Basic Research Foundation 2019B151530001, Hong Kong ITC ITF grants ITS/044/18FX and ITS/470/18FX, Microsoft Research Asia Collaborative Research Grant, HKUST-NAVER/LINE AI Lab, Didi-HKUST joint research lab, HKUST-Webank joint research lab grants.

REFERENCES

[1] Faisal N Abu-Khzam. 2010. A kernelization algorithm for d-hitting set. *J. Comput. System Sci.* 76, 7 (2010), 524–531.

[2] Christian Borgelt. 2012. Frequent item set mining. *Wiley interdisciplinary reviews: data mining and knowledge discovery* 2, 6 (2012), 437–456.

[3] B Barla Cambazoglu, Ismail Sengor Altingovde, Rifat Ozcan, and Özgür Ulusoy. 2012. Cache-based query processing for search engines. *ACM Transactions on the Web (TWEB)* 6, 4 (2012), 1–24.

[4] Jie Chen et al. 2018. FastGCN: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247* (2018).

[5] Jinyin Chen, Xuanheng Xu, Yangyang Wu, and Haibin Zheng. 2018. GC-lstm: Graph convolution embedded lstm for dynamic link prediction. *arXiv preprint arXiv:1812.04206* (2018).

[6] Zeyu Cui, Zekun Li, Shu Wu, Xiaoyu Zhang, Qiang Liu, Liang Wang, and Mengmeng Ai. 2021. DyGCN: Dynamic Graph Embedding with Graph Convolutional Network. *arXiv preprint arXiv:2104.02962* (2021).

[7] Ailin Deng and Bryan Hooi. 2021. Graph neural network-based anomaly detection in multivariate time series. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence, Vancouver, BC, Canada*. 2–9.

[8] Songgaojun Deng, Huzefa Rangwala, and Yue Ning. 2019. Learning dynamic context graphs for predicting social events. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1007–1016.

[9] Amit Kumar Dhar, Raghunath Reddy Madireddy, Supantha Pandit, and Jagpreet Singh. 2020. Maximum independent and disjoint coverage. *Journal of Combinatorial Optimization* (2020), 1–21.

[10] Johannes Gehrke, Paul Ginsparg, and Jon Kleinberg. 2003. Overview of the 2003 KDD Cup. *Acm Sigkdd Explorations Newsletter* 5, 2 (2003), 149–151.

[11] Palash Goyal, Nitin Kamra, Xinran He, and Yan Liu. 2018. Dyngem: Deep embedding method for dynamic graphs. *arXiv preprint arXiv:1805.11273* (2018).

[12] Gösta Grahne and Jianfei Zhu. 2005. Fast algorithms for frequent itemset mining using fp-trees. *IEEE transactions on knowledge and data engineering* 17, 10 (2005), 1347–1362.

[13] Dimitrios Gunopulos, Roni Khardon, Heikki Mannila, Sanjeev Saluja, Hannu Toivonen, and Ram Sewak Sharma. 2003. Discovering all most specific sentences. *ACM Transactions on Database Systems (TODS)* 28, 2 (2003), 140–174.

[14] Will Hamilton and thers. 2017. Inductive representation learning on large graphs. In *Advances in neural information processing systems*. 1024–1034.

[15] Yizeng Han, Gao Huang, Shiji Song, Le Yang, Honghui Wang, and Yulin Wang. 2021. Dynamic neural networks: A survey. *arXiv preprint arXiv:2102.04906* (2021).

[16] Dorit S Hochbaum. 1996. Approximating covering and packing problems: set cover, vertex cover, independent set, and related problems. In *Approximation algorithms for NP-hard problems*. 94–143.

[17] Yifan Hou, Hongzhi Chen, et al. 2019. A representation learning framework for property graphs. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 65–73.

[18] Wenbing Huang et al. 2018. Adaptive sampling towards fast graph representation learning. In *Advances in neural information processing systems*. 4558–4567.

[19] Zhihao Jia, Sina Lin, Rex Ying, Jiaxuan You, Jure Leskovec, et al. 2020. Redundancy-Free Computation for Graph Neural Networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 997–1005.

[20] Seyed Mehran Kazemi, Rishab Goel, Kshitij Jain, Ivan Kobyzev, Akshay Sethi, Peter Forsyth, and Pascal Poupart. 2020. Representation Learning for Dynamic Graphs: A Survey. *Journal of Machine Learning Research* 21, 70 (2020), 1–73.

[21] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[22] Boris Knyazev et al. 2019. Learning Temporal Attention in Dynamic Graphs with Bilinear Interactions. *arXiv preprint arXiv:1909.10367* (2019).

[23] Markus Kowarschik and Christian Weiß. 2003. An overview of cache optimization techniques and cache-aware numerical algorithms. *Algorithms for memory hierarchies* (2003), 213–232.

[24] Mantu Kumar, Neera Batra, and Hemant Aggarwal. 2012. Cache based query optimization approach in distributed database. *International Journal of Computer Science Issues (IJCSI)* 9, 6 (2012), 389.

[25] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*. 1343–1350.

[26] Ao Li, Zhou Qin, et al. 2019. Spam Review Detection with Graph Convolutional Networks. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. 2703–2711.

[27] Haoyang Li and Lei Chen. 2021. Cache-based GNN System for Dynamic Graphs Technique Report. https://drive.google.com/drive/folders/1BhhSFs7Fs0rahxWjeRLkb4BSxAcQvrzM?usp=sharing

[28] Liying Li, Guodong Zhao, and Rick S Blum. 2018. A survey of caching techniques in cellular networks: Research issues and challenges in content placement and delivery strategies. *IEEE Communications Surveys & Tutorials* 20, 3 (2018), 1710–1732.

[29] Wenjuan Luo, Han Zhang, Xiaodi Yang, Lin Bo, Xiaoqing Yang, Zang Li, Xiaohu Qie, and Jieping Ye. 2020. Dynamic Heterogeneous Graph Neural Network for Real-time Event Prediction. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3213–3223.

[30] Chen Ma, Liheng Ma, Yingxue Zhang, Jianing Sun, Xue Liu, and Mark Coates. 2020. Memory Augmented Graph Neural Networks for Sequential Recommendation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 5045–5052.

[31] Franco Manessi, Alessandro Rozza, and Mario Manzo. 2020. Dynamic graph convolutional networks. *Pattern Recognition* 97 (2020), 107000.

[32] Stefan Noll, Jens Teubner, Norman May, and Alexander Böhm. 2018. Accelerating concurrent workloads with CPU cache partitioning. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 437–448.

[33] Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, and Charles E Leisersen. 2019. Evolvegcn: Evolving graph convolutional networks for dynamic graphs. *arXiv preprint arXiv:1902.10191* (2019).

[34] Andrea Rossi, Denilson Barbosa, Donatella Firmani, Antonio Matinata, and Paolo Merialdo. 2021. Knowledge graph embedding for link prediction: A comparative analysis. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 15, 2 (2021), 1–49.

[35] Aravind Sankar, Yanhong Wu, Liang Gou, Wei Zhang, and Hao Yang. 2020. DySAT: Deep Neural Representation Learning on Dynamic Graphs via Self-Attention Networks. In *Proceedings of the 13th International Conference on Web Search and Data Mining*. 519–527.

[36] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. 2018. Self-attention with relative position representations. *arXiv preprint arXiv:1803.02155* (2018).

[37] Oleksandr Shchur and Stephan Günnemann. 2019. Overlapping community detection with graph neural networks. *arXiv preprint arXiv:1909.12201* (2019).

[38] Joakim Skarding, Bogdan Gabrys, and Katarzyna Musial. 2020. Foundations and modelling of dynamic networks using Dynamic Graph Neural Networks: A survey. *arXiv preprint arXiv:2005.07496* (2020).

[39] Rakshit Trivedi, Mehrdad Farajtabar, Prasenjeet Biswal, and Hongyuan Zha. 2018. Dyrep: Learning representations over dynamic graphs. (2018).

[40] Ruo-Chun Tzeng and Shan-Hung Wu. 2019. Distributed, Egocentric Representations of Graphs for Detecting Critical Structures. In *International Conference on Machine Learning*. 6354–6362.

[41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *arXiv preprint arXiv:1706.03762* (2017).

[42] Petar Veličković, Guillem Cucurull, Arantxa Casanova, et al. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[43] Le Wu, Peijie Sun, Richang Hong, Yanjie Fu, Xiting Wang, and Meng Wang. 2018. Socialgcn: An efficient graph convolutional network based model for social recommendation. *arXiv preprint arXiv:1811.02815* (2018).

[44] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826* (2018).

[45] Rex Ying, Ruining He, et al. 2018. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 974–983.

[46] Yong Yu, Xiaosheng Si, Changhua Hu, and Jianxun Zhang. 2019. A review of recurrent neural networks: LSTM cells and network architectures. *Neural computation* 31, 7 (2019), 1235–1270.

[47] Guangyi Zhang and Aristides Gionis. 2020. Diverse rule sets. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1532–1541.

[48] Muhan Zhang and Yixin Chen. 2018. Link prediction based on graph neural networks. *arXiv preprint arXiv:1802.09691* (2018).

[49] Yuan Zuo, Guannan Liu, Hao Lin, Jia Guo, Xiaoqian Hu, and Junjie Wu. 2018. Embedding temporal network via neighborhood formation. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2857–2866.