Representing Social Networks as Dynamic Heterogeneous Graphs

Negar Maleki  
University of South Florida, FL, USA  
negarmaleki@usf.edu  

Balaji Padmanabhan  
University of South Florida, FL, USA  
bp@usf.edu  

Kaushik Dutta  
University of South Florida, FL, USA  
duttak@usf.edu

Abstract—Graph representations for real-world social networks in the past have missed two important elements: (i) the multiplexity of connections and (ii) representing time. This paper presents a dynamic heterogeneous graph representation for social networks which includes time in every component of the graph, i.e., nodes and edges, each of different types that captures heterogeneity. We illustrate the power of this representation by presenting four time-dependent queries and deep learning problems that cannot easily be handled in conventional homogeneous graph representations. As a proof of concept we present a detailed representation of a relatively new social media platform (Steemit), which we use to illustrate both the dynamic querying capability as well as prediction tasks using Graph Neural Networks (GNNs). We also illustrate opportunities for future work in query optimization as well as new dynamic prediction tasks on heterogeneous graph structures.

Index Terms—Heterogeneous Graphs, Social Networks, Temporal Graphs, GNNs, Steemit

I. INTRODUCTION

Social media has become part and parcel of the everyday life of individuals. Applications such as Facebook, Snapchat, TikTok, YouTube, LinkedIn, Instagram, Twitter, and Reddit have proliferated worldwide; recent statistics suggest that over 10 billion hours a day are spent worldwide on social media alone. As users become more dependent on these platforms, their requirements naturally increase, and platforms need to continue to improve design and features to satisfy new needs.

As an example, querying social media to determine which users (an individual) may have interacted with in the last two years is a useful feature to help users maintain their relationships between users. Nodes and edges in the graphs will have attributes associated with entities in social media. To fully represent various entities in social media, such as ‘users’, ‘posts’ and ‘comments’ using graphs, a heterogeneous graph structure is needed [1]. The different relationships among these entities are also depicted using different types of edges such as ‘authored by’, ‘liked by’, and ‘commented by’. Hence, a representation that can allow both heterogeneities in types of nodes and edges is important.

Social media data is dynamic, and time plays a significant role in how users obtain value from these platforms. Thus, incorporating the time dimension into a heterogeneous social media graph is necessary. For instance, this can help create a sub-graph out of the full graph that can be used for focused analytical and machine learning tasks on social media data.

Still, the heterogeneous graph representing the full dynamic temporal aspect of social media is largely absent in the existing research. In this research, we present a dynamic heterogeneous graph to represent an upcoming social media, Steemit [2]. We then demonstrate the usefulness of this representation through the implementation of common social media queries and predictive analytic tasks.

A recent paper [3] has proposed a dynamic heterogeneous graph with temporal dimension for a similar purpose as described above. Authors of [3] implemented a graph neural network based on a heterogeneous temporal graph. In this, an individual heterogeneous graph is created at each $t$ and the relationship between the two graphs in consecutive time $t$ and $(t+1)$ is depicted through additional edges. Thus, if a node and an edge exist in both $t$ and $(t+1)$, the node and the edge will be created twice - once for graph $G_t$ at $t$ and once for graph $G_{(t+1)}$ at $(t+1)$. Additional edges will be created between nodes and edges in $G_t$ and $G_{(t+1)}$. In a very dynamic situation, such as in social media, where new nodes (depicting a post, a comment, or an user) are added quite frequently, the approach in [3] will create multiple heterogeneous graphs for each time interval. The representation in [3] will increase the effective number of nodes and edges in the graph. This will increase both the storage and computational complexity of using this representation in the context of social media data. Contrary to this approach, in our representation, we embed the time information in the nodes and edges themselves without replicating the same node and edges multiple times which makes the size of the graph manageable, and thus, can scale
better.

To show the expressive power of our proposed graph representation we use a social media platform, Steemit [2] as a case study. Steemit is a social media platform that runs on top of a blockchain called Steem, which rewards its users through a cryptocurrency, called Steem dollar. Steemit allows its members to publish content in the form of posts. Posts similar to Reddit are blogs users create and post to the Steemit website. Users can also up-vote/ down-vote other users’ posts, comment on posts, and resteem other users’ posts, which is similar to retweeting a post on Twitter or sharing someone else’s post on Facebook. Users can get paid for up-voting/ down-voting and commenting on other posts besides posting content.

In this paper, we first represent the Steemit data using the proposed dynamic heterogeneous graph structure. Next, we demonstrate how common time-dependent queries on social media can be efficiently answered using the proposed representation. Lastly, to demonstrate how GNN models can be adapted for our proposed graph representation we illustrate specific prediction tasks.

II. RELATED WORK

A. Graph Neural Networks

Graph data represent unique challenges for machine learning algorithms. Graph Neural Networks (GNNs) were developed for such data and had many designs dependent on how these define the concept of a message and the concept of aggregation (i.e. a node’s attributes are updated based on messages passed from neighbors). Graph Convolutional Network (GCN) [4] is a well-known GNN method that, in terms of weight sharing, is similar to a Convolutional Neural Network (CNN). Fast approximation spectral-based graph convolutional network [5] is one of the key algorithms classified as GCNs and utilized for node clustering. Kipf and Welling [5] built their approach by including the graph’s adjacency matrix in the forward propagation equation.

A limitation of the GCN algorithm is that it assigns the same level of importance to each edge. However, this problem is addressed in Veličković et al. [6] by expanding the aggregation function of the GCN layer by assigning attention coefficients to each edge. The aggregation function in the graph attention network (GAT) is similar to the one in GCN, except neighbors’ embeddings scaled by attention coefficients aggregate together. Algorithms that use low-dimensional node embeddings are used in different prediction tasks. However, most of these algorithms, like DeepWalk [7], are inherently transductive; adding a new node requires re-training the model, and cannot be generalized to unseen or new nodes. GraphSAGE is an algorithm that addresses this issue and can be used in dynamic graphs, where the structure is not fixed. GraphSAGE was first introduced by Hamilton et al. [8] to aggregate a new node nearby to generate node embedding for it efficiently. Therefore, we use GraphSAGE in our GNN architecture as the time component makes the graph dynamic.

B. Temporal Graphs

Temporal graphs were introduced in Kostakos [9], which represented a graph structure including time. The dynamic features of a dataset, as well as the entities and connections that it represents, can be understood using temporal graphs.

Most approaches focus on discrete-time dynamic graphs represented as a series of snapshots of the graph [10]–[13]; few techniques that support the continuous-time situation have recently been suggested. The Dynamic Graph Neural Network (DGNN) [14] model, used for link prediction and node classification, continuously updates node information by sequentially recording new edges (connections), the time window between edges, and information propagation. The continuous-time dynamic network embeddings (CTDNE) [15] algorithm learns embeddings based on the temporal random walks concept, which is used for link prediction. A temporal walk is a temporally valid sequence of edges walked in ascending order of edge timings that respects time. The system is broadly applicable, with many interchangeable components, and it can be used effectively to include temporal dependencies into current node embedding and deep network models that employ random walks. DyRep [16] is a novel modeling framework that learns richer node representation over time, which is used for link and time prediction. The Temporal Graph Attention (TGAT) [17] layer aggregates temporal and topological features besides learning time-feature interactions. By adding TGAT layers, the network learns node embeddings as a function of time and then can be used for node classification and link prediction tasks. Temporal Graph Network (TGN) [18] framework was developed at Twitter and can be applied to different problems represented as sequences of timed events.

All the above models are applicable to the homogeneous graph, where there are one type of node and one type of edge. In a recent paper [3], spatial and temporal dependencies of a heterogeneous graph were integrated to learn node representations over a Heterogeneous Temporal Graph (HTG) - described as an ordered list of heterogeneous graph slices connected by a set of temporal connections. They addressed a similar problem as we do in this paper (i.e. modeling heterogeneous graph with temporal dimensions), but with the difference that we have time in both edges and nodes of our heterogeneous graph instead of having them only for slicing the graph. To this end, like [3], we use time in slicing graphs for train-validation-test sets. Moreover, unlike previous graph representations, we include time in every component of the graph, i.e., nodes and edges, and use them in the GNN.

C. Multiplex Network

Multiplexity refers to multifaceted connections between two persons [19]; therefore, social networks are a good example of multiplex networks. A group of networks containing different types of relations is considered a multiplex network where each kind of relationship creates a layer of the network.

Real-world networks are significantly more complex than two-layer graphs, as cross-domain interactions across various networks are well-
observed, resulting in a sort of multi-layered network [20]. A recent paper studied how to perform network embedding for nodes on multiplex networks by providing a unified optimization framework, called MANE [20]. Their framework is considered as a heterogeneous information network, since different node types are placed in different layers. However, having different layers with different node types adds to the complexity of multiplex networks. Thus, researchers focus on different types of layers with a single node type and suggest a multiplex network embedding model, called MNE [21]. However, this framework aggregates information over all layers, even the irrelevant ones, so it is not computationally efficient. To address this drawback, the DEEPLEX framework was proposed, which considers sampling k-nearest layers with the most similar embeddings. In a recent paper [23], a framework is suggested to be more efficient in terms of computational complexity, which addressed the MNE problem by selecting information from relevant layers, and also addressed DEEPLEX drawback by adaptively learning a sampling distribution over the relevant layers. The idea of having information from relevant layers inspired us to extract sub-graphs out of the larger heterogeneous graph using time component queries. Although extracting sub-graphs decreased embedding and topological information, our findings show that we still get at least the same or better results than including all information in the payout classification problem, besides the obvious time efficiency.

III. GRAPH REPRESENTATION

A graph is a generic mathematical language for describing complex networks. Graphs are commonly represented as \( G = (V, E) \), with nodes \( V \) and edges \( E \). In this paper, the graph is constituted of actors and actionable items < \( \text{actor}_1 \rightarrow \text{action} \rightarrow \text{actor}_2 \) >, where each may have its own set of attributes. Both the actor and the actionable item have time attributes associated with them. For actors, the time indicates when the actor was created (such as ‘joined’ for users and ‘created’ for posts and comments). The actionable item has a time attribute indicating when the action (such as authored or voted) took place.

Attributes can be divided into two main categories, static and dynamic attributes. Static attributes mainly define the state of an actor or actionable item (such as ‘permalink’ and ‘category’). However, dynamic attributes, as their name states, can be changed over time, so they are time-dependent attributes (such as the ‘payout’, ‘author_reputation’, and ‘author_rewards’). Networks, in reality, are not static, which means actors take action and modify the networks in various ways.

Three main operations take place in networks: item addition, item deletion, and item modification. In item addition operation, new actors or actionable items appear and create new connections to the network, which extend the network. Item modification operation gives actors or actionable items an opportunity to update or modify the existing actors or actionable items. Lastly, the item deletion operation removes existing actors or actionable items where all the dependent children will disappear as their parents are removed.

Graphs are searchable and enable us to query and extract intuitive information, which later can be used in different ML tasks. Relational algebra operations like selection, projection, and join are common among different database management systems (DBMS), including network database systems. Selection (\( \sigma \)) operation as one of the unary relational operations is used for selecting a subset of tuples that satisfy the selection condition. Projection (\( \pi \)) operation as another operation in the unary relational operations is used for keeping specific attributes from a relation and ignoring the rest of attributes. Lastly, the join (\( \bowtie \)) operation as one of the binary relational operations is used for joining variously related tuples from different relations.

We followed the representation in [1] to illustrate Steemit in form of a graph in Fig 1. However, we have adapted it to the actions in Steemit (such as vote vs. like) and added the time attribute in the graph in Fig 1.

| Item       | Item Type | Description |
|------------|-----------|-------------|
| authored   | Actionable| A person is the author of a post. |
| reply      | Actionable| A post is replied by a comment. |
| vote       | Actionable| A post is voted by a person. |
| follow     | Actionable| A person is followed by a person. |
| user       | Actor     | Take actionable items on another actor. |
| post       | Actor     | Take actionable items on another actor. |
| comment    | Actor     | Take actionable items on another actor. |

A. Dataset

For demonstration purposes, we use the SteemOps data from Steemit in this paper. SteemOps is a dataset presented in [24], which contains ten key types of Steemit operations organized into three sub-datasets: (1) the social-network operation dataset (SOD), (2) the witness-election operation dataset (WOD), and (3) the value-transfer operation dataset (VOD). The data was collected from 2016/03/24 16:05:00 to 2019/12/01 00:00:00. The main sub-dataset we use in this paper is SOD consisting of three operational keys - comment, vote, and custom-json. The comment operation consists of five fields, see Table II. According to [24], a new post is indicated when both parent-author and parent-permalink fields are empty. When these two fields are not empty, it represents a comment to a post/comment. Each post in Steemit remains active for 7-day, so each time the author makes any changes to his post, the post will be recorded as a new post to the dataset. Moreover, each post’s permlink is unique, so considering all of these, the dataset consists of 17805355 new posts.

The Steem platform offers an interactive Application Programming Interface (API) for researchers to parse the data. However, just retrieving the full information considering some API restrictions, would have taken approximately 38 days in
TABLE II
SCHEMA OF OPERATION COMMENT [24]

| Field name | Type     | Description                           |
|------------|----------|---------------------------------------|
| block_no   | Integer  | the block recording this operation    |
| parent_author | String  | the author that comment is being submitted to |
| parent_permlink | String  | specific post that comment is being submitted to |
| author     | String   | author of the post/comment being submitted (account name) |
| permlink   | String   | unique string identifier for the post, linked to the author of the post |

In this dataset, we retrieved a random 10% of this dataset (approximately 1.7 million new posts) for demonstration purposes in this paper. For these 1.7 million posts, we also retrieved relative information like comments on the posts and voters. We randomly selected half of the August 2019 data and converted it into a graph to demonstrate in this paper. The entire heterogeneous graph contains 165180 nodes and 1405939 edges after construction, which covers 24000 posts published in August 2019.

B. Heterogeneous Graph Construction

We represent Steemit as a directed graph \( \langle V, E \rangle \), where \( V \) represents actors (nodes), and \( E \) represents actionable items (edges). In Steemit, an actor can be of different types, such as user and content (post or comment). The interactions between actors are captured by actionable items, which also can be of different types, see Table I. For example, a user can follow another person, which we denote as \( <user_1 \rightarrow follow \rightarrow user_2> \in E \). Similarly, a user can make a post, which we denote as \( <user_1 \rightarrow Authored \rightarrow post_1> \in E \), or a user can vote a post or a comment, which we denote as \( \{<user_1 \rightarrow vote \rightarrow post_1>, <user_1 \rightarrow vote \rightarrow comment_1>\} \subset E \).

The heterogeneity of the graph is indicated through the attribute ‘otype’ in the nodes indicating actors, which can be ‘user’, ‘comment’, or ‘post’. Similarly, the edges indicating actionable items can be of different types ‘follow’, ‘vote’, ‘authored’, and ‘reply’. As mentioned before, actors and actionable items have static and/or dynamic attributes. For example, user object type (otype) or actor has “name”, “bot”, and “joined” date as static attributes and the number of followers and following as dynamic attributes that can be calculated using graph queries. The “post” object type (otype) or actor has many static attributes illustrated in Fig. 1, except “title”, “body”, and “last_update” which are considered as dynamic attributes where their changes will be stored in form of a list or JSON depending on the attribute characteristics, and also a total number of post’s votes can be calculated using graph queries (sum of incoming votes (actionable item)). Note that the dynamic attributes calculated through queries do not need to be explicit in the graph representation. Both the actions (nodes) and the actionable items (edges) have time as one attribute, which makes it a Dynamic Heterogeneous Graph.

We constructed the Steemit graph as a directed graph (DiGraph) using the NetworkX Python library. The constructed graph is then converted into the PyTorch Geometric (PyG) graph in order to be used for GNN models. By doing so, we create a dynamic graph from which needed sub-graphs can be extracted using queries, and the derived sub-graphs may subsequently be utilized in various deep learning tasks.

We demonstrate the utility of this heterogeneous dynamic graph representation of Steemit with the help of common queries and two applications of GNN machine learning on this graph representation.

IV. GRAPH QUERIES

The entire graph is not always the point of interest in different tasks to be performed on social networks, such as the Steemit graph. The unique aspect of Steemit graph representation is the time attribute in both actions and actionable items. To this end, we may like to take a sub-graph out of the entire graph, which represents all activities between a specific time interval. The application of such a query could be manyfold; for example, in presidential election seasons, people’s activities on social media become attractive. Therefore, analysis of such activities gives interesting information...
required to be gained from specific time intervals. The key point in this representation is that we only include actions and actionable items relevant in the specified time intervals. Any actions or actionable items connected to nodes but did not occur during the given time interval will be removed from the sub-graph. Algorithm 1 shows the pseudo-code of the sub-graph query between $t_1$ and $t_2$, which is the first portion for all the subsequent queries discussed in this section. In the following queries, the output of the Algorithm 1 will be given as an input. The time complexity of the Algorithm 1 is $O(n + m)$ where $n$ is the number of nodes, and $m$ is the number of edges.

Algorithm 1 Query Sub-Graph between $t_1$ and $t_2$

1: Input: $G(V,E)$: graph with $n$ nodes and $m$ edges
2: Input: $t_1, t_2$: timestamp
3: Output: $G_{post}(t_1,t_2)$: Subgraph between $t_1$ and $t_2$
4: define function get_key(val):
5: set key_idx_node to []
6: set key_idx_edge to []
7: for key, value in networkX.get_attributes(G, 'otype').items() do
8: if val equals value then
9: add key to key_idx_node
10: add key to key_idx_edge
11: end if
12: end for
13: return key_idx_node, key_idx_edge
14: call set post, comment, user to get_key(‘post’, ‘comment’, ‘user’)
15: call set vote, authored, reply to get_key(‘vote’, ‘authored’, ‘reply’)
16: set subG_post, subG_comment to []
17: for i in post (or comment) do
18: if $t_1$ ≤ G.nodes()(i)’ ‘created’ ≤ $t_2$ then
19: add i to subG_post (or subG_comment)
20: end if
21: end for
22: set subG_vote, subG_authored, subG_reply to []
23: for i in vote (or authored or reply) do
24: if $t_1$ ≤ G.edges()(i)’ ‘time’ ≤ $t_2$ then
25: add i to subG_vote (or subG_authored or subG_reply)
26: end if
27: end for
28: set G_node to G.subgraph(subG_post + subG_comment + user)
29: set G_post/t1,t2 to G_node.edge_subgraph(subG_vote + subG_authored + subG_reply)

Consider Fig. 2, which shows a heterogeneous network including users, posts, and comments as nodes and edges connecting these nodes (node types and edge types are as described in Table 1). Along with each edge, the $t$ shows when the actionable items took place, and the subscripts show the order of those actionable items, such as $t_2 > t_1$. Fig. 2 shows the result of Algorithm 1 if we are interested in constructing the sub-graph between $t_1$ and $t_6$. All the nodes and edges that are not placed in the given time interval will be deleted and the final sub-graph will include exactly actions and actionable items between $t_1$ and $t_6$.

To show the expressive power of time in the Steemit graph representation, the sub-graph extracted for a given time interval will be used as the input of the following queries. We present three different queries that have common applications in the social media context.

A. Querying on Category of Posts

The category of a post is one of the attributes in Steemit. This attribute can be used for selecting posts in a specific field. For example, for health-related posts, we may look for categories such as 'healthcare' or 'medicine’. Algorithm 2 is the query following the sub-graph between $t_1$ and $t_2$ to select posts with a specific category within that time frame. The time complexity of this algorithm is $O(n^2)$ where $n$ is the number of nodes in sub-graph between $t_1$ and $t_2$.

Consider Fig. 3, which shows a heterogeneous network with nodes representing users, posts, and comments. The goal is to have specific posts in a specific category within a given time interval. Posts’ categories are $A, B, C, D$, and we are interested in $C$ and $D$ categories between $t_1$ and $t_6$. After applying Algorithm 2 to this graph, any nodes and edges that are not placed in the given time interval and any posts that do not belong to $C$ and $D$ will be eliminated. Therefore, the final sub-graph will include actions and actionable items between $t_1$ and $t_6$, as well as posts with $C$ and $D$ categories only.

B. Popularity Query

The two most important factors for determining how much payout posts get (denoting popularity) in Steemit [24] are the number of votes and comments on a post. Thus to identify the posts that will have the higher payouts (or are more popular), we may sort posts based on the number of votes and comments. Algorithm 3 is the query on the sub-graph between $t_1$ and $t_2$ to sort posts based on their vote and comment count. Its time complexity is $O(nlog(n) + nm)$, where $n$ is the number of
Algorithm 2 Query SubGraph between $t_1$ and $t_2$ with an specified category

1: **Input:** $G_{post_{t_1,t_2}}(A,B)$ : Subgraph between $t_1$ and $t_2$
2: **Input:** category: string
3: **Output:** $G_{post_{t_1,t_2 category}}(C,D)$ : Subgraph between $t_1$ and $t_2$ with an specified category
4: call set post to get_key(‘post’)
5: set subG to post
6: for i in post do
7: if $G_{post_{t_1,t_2}}\textnodes(i)[’category’]$ equals category then
8: add i to subG
9: end if
10: end for
11: set related_node to []
12: set related_cm to []
13: for i in subG do
14: for j in range(len(subG_vote)) do
15: if i equals subG vote[j][1] then
16: add subG vote[j][0] related_node
17: end if
18: end for
19: end for
20: for i in related_cm do
21: for j in range(len(subG vote)) do
22: if i equals subG vote[j][1] then
23: add subG vote[j][0] related_node
24: end if
25: end for
26: end for
27: end for
28: end if
29: end for
30: end for
31: for i in related cm do
32: for j in range(len(subG vote)) do
33: if i equals subG vote[j][1] then
34: add subG vote[j][0] related node
35: end if
36: end for
37: end if
38: end for
39: add subG authored[j][1] related_node
40: end if
41: end for
42: end for
43: set category node to drop duplicates(related node)
44: set $G_{post_{t_1,t_2 category}}$ to $G_{post_{t_1,t_2} \textsubgraph (category \textnode + \textsubgraph_{t_1,t_2} \textpost)}$

Algorithm 3 Query to Sort posts based on their vote and comment count

1: **Input:** $G_{post_{t_1,t_2}}(A,B)$ : Subgraph between $t_1$ and $t_2$
2: **Output:** comments_sorted, votes_sorted: Lists of posts in descending order based on their vote and vote count
3: set vote_count to [1]
4: set comment_count to [1]
5: for i in $G_{post_{t_1,t_2}}\textnodes()$ do
6: set vote to 0
7: set comment to 0
8: if $G_{post_{t_1,t_2}}\textnodes(i)[’otype’]$ equals ‘post’ then
9: for j in $G_{post_{t_1,t_2} in\textedges()}$ do
10: if $G_{post_{t_1,t_2} in\textedges()}$ then
11: comment += 1
12: end if
13: end for
14: add [i, comment] to comment_count
15: for j in $G_{post_{t_1,t_2} in\textedges()}$ do
16: if $G_{post_{t_1,t_2} in\textedges()}$ then
17: vote += 1
18: end if
19: end for
20: add [i, vote+1] to vote_count
21: end if
22: end for
23: set comments_sorted to sorted(comment_count, key=lambda x: x[1], reverse to TRUE)
24: set votes_sorted to sorted(vote_count, key=lambda x: x[1], reverse to TRUE)

As an example of the larger Steemit graph, consider that we are interested in sorting larger posts based on their votes and comments count between "Thursday, August 1, 2019, 4:00:00 AM GMT" and "Saturday, August 10, 2019, 4:00:00 AM" (10-day). In the entire heterogeneous graph, we have 165180 nodes and 1405939 edges. Out of this, within the specified time interval, the number of nodes and edges are 85062 and 907437, respectively. Table III includes 10 top posts’ titles and payout with their number of votes and comments within the given time interval. The total execution time of this query was

Algorithm 2: Sub-graph between $t_1$ and $t_2$ with C and D categories in posts

Fig. 3. Result of the Algorithm 2: sub-graph between $t_1$ and $t_2$ with C and D categories in posts
C. Actively Engaged Query

Social media platforms quickly became the most used online services, through which billions of users interact intensively every day. This makes social media a significant resource for advertising, marketing, politics, collecting information, and launching campaigns. A challenging problem is identifying actively engaged users on Steemit (or any social media), where an actively engaged user is described as an user with a higher number of votes, comments, and the number of posts they have published. Algorithm 4 is the query on the sub-graph between \( t_1 \) and \( t_2 \) to sort users based on their activities within that time interval. Its time complexity is \( O(n) \), where \( n \) is the number of nodes in sub-graph between \( t_1 \) and \( t_2 \).

\[
\text{Algorithm 4 Query to Sort users based on Activities}
\]

1: Input: \( G_{\text{post}} \cdot t_1 \cdot t_2\) (A, B) : Subgraph between \( t_1 \) and \( t_2 \)
2: Output: user activity sorted: List of users in descending order who are actively engaged
3: set user activity to []
4: for \( i \) in range(len(user)) do
5: if user[i] in \( G_{\text{post}} \cdot t_1 \cdot t_2\).nodes() then
6: add (user[i], len(G_post \cdot t_1 \cdot t_2.out_edges(user[i]))) to user activity
7: end if
8: end for
9: set user activity sorted to sorted(user activity, key=lambda x: x[1], reverse to TRUE)

As another example, consider we are interested in knowing users who are actively engaged in the Steemit platform between "Thursday, August 1, 2019, 4:00:00 AM GMT" and "Saturday, August 10, 2019, 4:00:00 AM". Table IV includes 10 top users with the number of activities in casting votes and writing posts or comments. The total execution time of this query was 14.1105 seconds; Seconds on a 2.7GHz processor with 16GB RAM. (Note that in Steemit, usernames are anonymous but publicly available.)

D. Query Performance

As we discussed in the previous sub-sections, these queries have a low execution time compared to the size of the graph (some examples illustrated run times on a single machine; here we present query performance results on a compute cluster). Fig. 4 illustrates the execution time of each query for a different number of an initial set of posts on 1 node with 128 AMD Epyc 7702 CPU @ 2.0GHz processors and 1 TB memory. We focused on the number of posts in each run and implemented four queries for those posts. We considered the same \( t_1 \) and \( t_2 \) for all the queries. For the second query, we retrieved posts with the "Steemit" category. As demonstrated in Fig. 4, the query execution time even for 10K posts is below 10 seconds. However, we note that the focus of this paper is a new representation of social network data that combines dynamic and heterogeneous graphs, and shows the value of this representation through illustrative dynamic queries and prediction tasks. We have not addressed the issues of query optimization for real-time performance, which is an important topic for future work.

V. ILLUSTRATIVE PREDICTION TASKS

In this section, we show how the dynamic heterogeneous network representation can also be used in illustrative prediction tasks. Predicting the payout for a newly published post is one of the most important classification problems in the Steemit setting. Being able to predict the financial reward from a post can be used in ranking algorithms for content and can also be used as a tool to help users develop better content to post on social media. This problem is also equivalent to predicting whether a post on traditional social media will gain popularity in terms of the number of comments and likes received.
According to the Steemit documentation, the payout is determined by a variety of factors, including the number of up-votes and down-votes, the number of comments, the reputation scores of users who voted or replied to the post, how much new Steem Dollar (the cryptocurrency in Steemit platform) is created each week and distributed to users as rewards over a rolling 7-day period. As there are so many dynamic elements that impact the ultimate payout, no exact formula defines what the payout of a new post will be. As shown in Table III, posts with a high number of comments and votes did not necessarily get paid highly. Although predicting payout seems to be a regression task, the unavailability of time series data on payouts limited us to define this problem as a classification task. We have defined this problem as a classification problem that classified the payout amount of a Steemit post into low, medium, and high categories. Fig. 5 shows the distribution of the number of posts with non-zero payout. Here, we use this prediction problem to illustrate the kind of deep learning approach that can be applied in our dynamic heterogeneous graph representation of social network data.

### A. Predicting Payout of a Post using Post Features

In this approach, we include all the published features in the dataset (see Table V) for each type of node and edge, along with topological information of the graph, and then predict the payout of a new post. As we have time in each component of the graph, we ensure using queries discussed earlier that all the connections to each post node are in a 7-day time window (payouts occur 7-days after a post is made). Moreover, we divide the entire graph into three sub-graphs for train, validation, and test with %80, %10, and %10 of heterogeneous graph data, respectively, which would be used as input into the GNN model, similar to [3].

As we mentioned in Section III-B, we build the heterogeneous graph using the NetworkX Python library, and for implementing GNN models, we convert it into the form needed for the PyTorch Geometric (PyG) library. However, so far, there is no built-in function in the PyTorch library to convert a heterogeneous graph into PyG format. Therefore, this example also presents some insights to convert heterogeneous NetworkX graph into a heterogeneous PyG graph.

Heterogeneous graphs have different attributes within nodes and edges. Therefore, each node or edge has its own attributes with various dimensionality. As a result of the heterogeneous data structure, standard Massage Passing (MP) in GNN cannot be easily applied. In the heterogeneous graph models, during message computation, the MP algorithm should run over edge dictionaries, and during node updates, it should iterate over node-type dictionaries.

As shown in Fig. 6, the method starts with a GAT-GNN model and repeats the message functions to operate separately on each edge type. As discussed earlier, we have three node types: user, post, and comment, and each has its own tensor data. At first, we normalized all tensor data and then entered them into the GAT convolution layer (each node type has its own layer) to aggregate temporal and topological features. The outputs of GAT layers are new sets of node features given to linear transformation to obtain sufficient expressive power of higher-level features. We then use the ReLU activation function to concatenate all the features and repeat this process one more time.

1) Result: We split the graph into three sub-graphs – train contains 19200 posts, validation contains 2400 posts, and test contains 2400 posts. We trained the model using 128 hidden layers, and 5000 epochs with a 0.001 learning rate. The model shows %98.44, %90.19, and %87.44 accuracies for train, validation, and test sub-graphs, respectively. Table VI

### Table V

| Features        | Node Type          | Description                      |
|-----------------|--------------------|----------------------------------|
| node_type       | post, user, comment| Type of node                     |
| net_rshares     | post, comment      | Sum of positive and negative rewards |
| abs_rshares     | post, comment      | Total absolute weight of votes    |
| vote_rshares    | post, comment      | Total positive rshares from all votes |
| author_rewards  | post, comment      | Tracks the author payout          |
| author_reputation| post, comment      | Author’s reputation               |
shows the performance criteria and confusion matrix of the model.

**TABLE VI**

| Performance Criteria | Confusion Matrix |
|----------------------|------------------|
|                      | Precision Recall F1-score | Low | Medium | High |
| Low                  | 0.99 0.88 0.93 | 1964 | 277 | 0 |
| Medium               | 0.31 0.84 0.46 | 11 | 126 | 13 |
| High                 | 0.28 1.00 0.43 | 0 | 0 | 5 |

2) Sensitivity Analysis: As we have an unbalanced dataset (i.e., non-zero payouts form %30 of the dataset), we decided to assign a separate class to zeros, and change the prediction problem from 3 classes into 4 classes to see whether the model performance changes. We use the same hyper-parameters as we had for 3 classes, and we got %91.34, %80.93, and %79.38 accuracies for train, validation, and test sub-graphs, respectively. Table VII shows performance criteria and confusion matrix for the model. The accuracy of the 4-class problem reduces from the 3-class problem. This indicates that the result of the 3-class problem was biased by an unbalanced dataset.

**TABLE VII**

| Performance Criteria | Confusion Matrix |
|----------------------|------------------|
|                      | Precision Recall F1-score | Zeros | Low | Medium | High |
|                      | 0.98 0.87 0.92 | 1575 | 235 | 0 | 0 |
| Low                  | 0.45 0.45 0.45 | 17 | 195 | 219 | 0 |
| Medium               | 0.37 0.85 0.51 | 8 | 3 | 127 | 12 |
| High                 | 0.29 1.00 0.45 | 0 | 0 | 0 | 5 |

B. Predicting Payout of a Post using Post Content

In the second approach, we take the embedding vectors out of the body of the posts using a pre-trained Google Universal Sentence Encoder model. The model encodes text into 512-dimensional vectors ready to use for text classification, clustering, semantic similarity, etc. The pre-trained model is publicly available in Tensorflow-hub [1]. It is available in two variants, one trained using a Transformer encoder and the other with a Deep Averaging Network (DAN). There are trade-offs between these two variants’ accuracy and computational resource requirements. The Transformer encoder is more accurate, despite its high computational cost, while the DNA encoder is less costly computational, despite its lower accuracy [25]. To this end, we used the one with a transformer to encode the posts’ content into 512-dimensional vectors and gave them to the GNN model to predict new post payout.

Again, as we have the heterogeneous graph, different types of node and edge features exist. Thus, the standard MP cannot easily be applied and processed by the same function. As shown in Fig. 7, the method starts with a SAGE-GNN model and repeats the message functions to operate separately on each edge type.

**TABLE VIII**

| Performance Criteria | Confusion Matrix |
|----------------------|------------------|
|                      | Precision Recall F1-score | Zeros | Low | Medium | High |
|                      | 0.98 0.87 0.92 | 1575 | 235 | 0 | 0 |
| Low                  | 0.45 0.45 0.45 | 17 | 195 | 219 | 0 |
| Medium               | 0.37 0.85 0.51 | 8 | 3 | 127 | 12 |
| High                 | 0.29 1.00 0.45 | 0 | 0 | 0 | 5 |
VI. CONCLUSION

In this paper, we presented a new approach to representing social networks as dynamic heterogeneous graphs. We showed how this representation handles the inherent multiplexity that is part of real-world social networks, while also representing time explicitly in a way that enables powerful querying and predictive tasks. We presented several examples using a real-world heterogeneous social network, Steemit, where there are monetary incentives for users as well. We demonstrated how common queries on the Steemit social media graph could be executed efficiently using this graph structure. We also demonstrate how deep learning-based predictive models can be applied using this graph structure on the Steemit data. As far as we know, this is one of the first papers to have modeled real-world social networks using both time and heterogeneous graph representations.

There are many applications as well as important opportunities for future work. It is well-known that representation - and how data is stored in the back end of a system - drives functionalities that can be provided on the front end to users. Representations such as the one presented here can more easily enable a range of new user-functionalities in social media platforms that can allow users to query items (e.g., other users, or content types such as posts, videos, comments, etc.) based on temporal constraints.

Further, if deep learning on graphs extends to dynamic heterogeneous graphs, it can allow many new predictive models to also be developed on such data. These models may, for example, in real-time, be able to predict how much engagement a specific user’s post or comment may receive in a short span of future time. Such predictions can help platforms improve ranking algorithms and can also be used to offer users suggestions on how to develop content that can be more engaging. This is important in platforms such as TikTok, where there is an entire ecosystem of content creators who depend on the engagement of content for their livelihoods.

However, while our work here offers a potentially more expressive representation of social media data, there are challenges that need to be addressed in future work to make the functionalities mentioned previously practical. These include work on query optimization that can scale queries on dynamic heterogeneous graphs such that these can be executed in real time. Also, there are many opportunities to design better deep learning algorithms specifically tailored for dynamic heterogeneous graphs. Further, there might be alternate formulations of heterogeneous graphs that offer similar benefits within more constrained applications or functionalities. These are all interesting directions for future work.

REFERENCES

[1] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dumov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, and M. Marchukov, “TAO-Facebook’s Distributed Data Store for the Social Graph,” In2013 USENIX Annual Technical Conference, 2013, pp. 49-60.

[2] U. W. Chohan, “The Concept and Criticisms of Steemit,” (February 20, 2021), CBRI Working Papers: Notes on the 21st Century, Available at SSRN: 10.2139/ssrn.3129410.